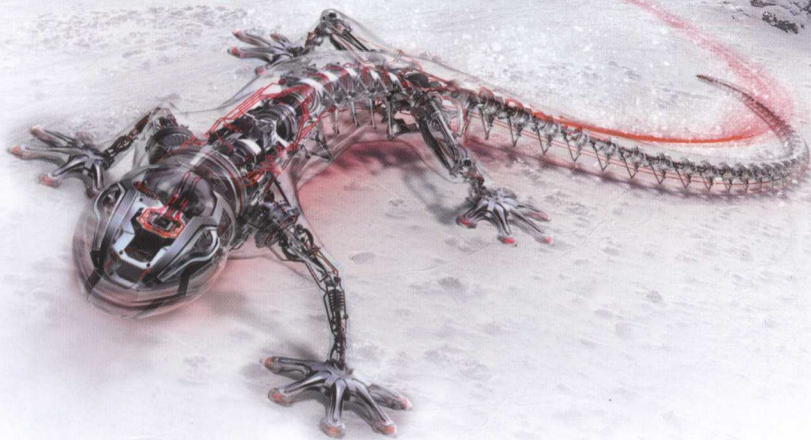


## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



针对Java游戏服务器技术进行深入剖析，为你层层揭开游戏服务器的神秘面纱。



# 深度解析 Java 游戏服务器开发

何金成 ◎ 编著

**范例应用：**提供30个可使用的编程实例与一个完整项目案例

**多维视角：**4大服务器开发技术讲解，6大线上游戏架构解析

**由浅入深：**清晰讲解40多个游戏服务器开发技术点

**案例分析：**RTS卡牌手游《皇室战争》开发实战



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

## 作者简介

### 何金成

游戏行业开发者，从事游戏服务器研发工作，曾就职于游戏谷，参与大型3DMMORPG《七雄无双》的服务端研发，后任职熊孩子游戏，并担任Java服务端主程，负责《英雄守卫战》、《王者守卫战》、《国战三国志》及《英雄对冲》等多款手游的服务端研发工作。腾讯GAD开发社区特约撰稿人。涉猎广泛，擅长Java后端研发，分布式服务开发，同时也精通Cocos、Egret等游戏前端引擎，曾在博客写过很多详细的游戏开发案例，掌握多种技术，并且非常乐于分享，是一位有着游戏梦的游戏开发者。

# 深度解析

## Java 游戏服务器开发

何金成 ©编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

## 内 容 简 介

为了帮助想了解入门游戏服务器开发的从业人员或非从业人员迅速掌握 Java 游戏服务器开发的技术, 本书从游戏的行业分析、Java 技术、游戏逻辑、数据库技术、网络理论、服务器技术、架构分析、系统优化等方面对游戏服务器开发做了全面解析, 并对目前市面较热门的游戏进行分析, 从行业到理论到技术, 再到架构到实战。本书带领读者熟悉 Java 游戏服务器开发相关领域, 帮助想要入门游戏服务器领域的读者更快地了解并掌握相关内容。

本书实用性强, 既是非游戏行业人员迅速了解并掌握游戏服务器技术的宝典, 又是游戏行业从业人员进阶提升的实用手册。

本书适合作为非游戏行业但想入门游戏行业的 Java 工程师、想了解游戏服务端技术的游戏前端工程师、需要游戏服务器开发入门工具书的人员, 以及其他对游戏服务器开发有兴趣爱好的人员的阅读书籍。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有, 侵权必究。

## 图书在版编目 (CIP) 数据

深度解析 Java 游戏服务器开发 / 何金成编著. —北京: 电子工业出版社, 2017.1

ISBN 978-7-121-30142-1

I. ①深… II. ①何… III. ①游戏—网络服务器—JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字 (2016) 第 251755 号

策划编辑: 李 冰

责任编辑: 李 冰

特约编辑: 田学清 赵海军等

印 刷: 北京中新伟业印刷有限公司

装 订: 北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×1092 1/16 印张: 24.5 字数: 510 千字

版 次: 2017 年 1 月第 1 版

印 次: 2017 年 1 月第 1 次印刷

定 价: 79.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式: [libing@phei.com.cn](mailto:libing@phei.com.cn)。



# 前言

## PREFACE

Java 诞生于 20 世纪 90 年代，是一款面向对象的工具，为企业级的计算领域提供解决方案。目前，Java 普遍应用于分布式开发及大数据云计算领域，无论是大家手中的安卓手机，还是企业级 CS、BS 项目，以及微信公众号和 APP 后台开发，处处可见 Java 的身影，而本书将重点介绍 Java 在游戏服务器开发领域的应用。

### 本书特色

本书将讲解在游戏开发领域，如何使用 Java 开发游戏服务器。本书附有丰富的源码案例，并对目前较热门的网络游戏服务器进行分析。

市面上大部分讲解 Java 的书都以 Java 基础、JavaEE 企业级开发、Android 开发、微信开发等为主，本书将重点讲解 Java 在游戏领域的应用——Java 作为游戏服务器开发的卓越表现。虽然，服务器领域一直由 C++ 主导，C++ 优秀的性能在服务器中有着良好的表现，但近些年来，Java 也变得越来越优秀，Java 在网络方面的处理性能也有了显著提升，使用 Java 开发游戏服务器也成为可能。本书为使用 Java 开发游戏服务器提供指导，让有 Java 基础的读者快速入门 Java 游戏服务器开发。

本书对游戏服务器开发的各种技术都做了详细介绍，并且源码中有相应的案例，代码中也有非常详细的注释。对于不容易理解的内容，结合图表进行讲解，使读者可以有一个更加直观的认识。

书中涉及的均是目前使用较广泛、较热门、较实用的技术，想要了解新技术的人可以把本书作为参考。

## 本书优势

虽然目前也有少量关于游戏服务器的书讲解得非常细致，但大多是基于 C++ 语言实现的服务端内容，而使用 Java 开发游戏服务器的开发商越来越多，本书针对的就是有一定 Java 基础的人，每章均配有源码案例，可带领读者快速入门 Java 游戏服务器开发。

## 为什么是 Java?

JDK4 提供了 NIO 类库——异步 IO，NIO 框架也越来越被人熟知，而且 Mina、Netty 都是基于 NIO 网络框架的。随着大规模分布式系统、大数据和流式计算框架的崛起，基于 Java 构建这些系统已经成为主流，NIO 编程和 NIO 框架也得到了广泛应用。在互联网领域，阿里的分布式服务框架 Dubbo、RocketMQ、大数据的基础序列化通信框架 Avro，以及很多开源的软件都已经使用 Netty 框架构建高性能、分布式的通信。所以，Java 是能胜任游戏服务器开发的，也完全能满足当下的游戏性能要求。

## 本书各章内容

本书分为基础篇、入门篇和高级篇三个篇章，共 10 章。

### 基础篇

从游戏行业的基础知识讲起，带领读者进入游戏开发领域，做一些前置的准备工作。

### 第 1 章

讲解游戏的理论知识、游戏行业的现状及发展、游戏开发中的要点等，让读者了解游戏服务器开发的基本内容。

## 第2章

为后续开发做准备，讲解 Java 开发环境搭建，并介绍了一些常用开发工具。

### 入门篇

以 Java 游戏服务器技术为核心，分模块展开讲解，真正从技术上了解并掌握游戏服务器开发技术。

## 第3章

网络对游戏服务器起着至关重要的作用，本章讲解游戏服务器开发中的网络层。

## 第4章

传输数据时，需要拟定双方都能解析的数据格式，使双方都能辨别。

## 第5章

游戏数据多种多样，如何进行数据缓存，如何使数据持久化，本章将一一进行介绍。

## 第6章

游戏逻辑是游戏服务器的核心，不同游戏的业务逻辑，对应着不同的逻辑代码。本章将介绍如何根据游戏业务逻辑进行逻辑层的开发。

## 第7章

开发游戏服务器时必须要做安全性保护。在外挂满天飞的时代，一个优秀的游戏服务器必须对各种有意或无意的攻击做好安全防护。

### 高级篇

讲解游戏服务器的架构设计及系统优化，从更高的角度了解游戏服务器，并用一个实例代码进行具体说明。

## 第8章

分析市面上常见网络游戏的服务器架构，学会分析优秀的架构，了解好的架构对游戏服务器的重要性。

## 第9章

本章讲解一款游戏实战，指导读者上手开发一款游戏服务器，了解游戏服务器开发

中的基本模式，也是对本书所讲内容的一个综合应用。

## 第 10 章

本章展望游戏行业的未来，分析游戏行业的走向。

## 本书读者对象

- ▶ 非游戏行业但想入门游戏行业的 Java 工程师。
- ▶ 想了解游戏服务端技术的游戏前端工程师。
- ▶ 需要游戏服务器开发入门工具书的人员。
- ▶ 其他对游戏服务器开发有兴趣爱好的各类人员。



# 目录

## CONTENTS

### 基础篇 走进游戏开发

第 1 章 认识游戏 .....	2
1.1 什么是游戏 .....	2
1.1.1 游戏的定义 .....	2
1.1.2 游戏的分类 .....	3
1.2 游戏开发及分工 .....	7
1.3 游戏行业现状分析 .....	12
1.4 游戏服务器开发要点 .....	15
1.5 总结 .....	17
第 2 章 环境搭建 .....	18
2.1 Windows 开发环境搭建 .....	18
2.1.1 安装 JDK .....	18
2.1.2 安装 Eclipse .....	20
2.1.3 安装数据库客户端工具 .....	21
2.1.4 安装 SSH 工具 .....	21

2.1.5 安装其他工具.....	22
2.2 Mac OS X 开发环境搭建.....	22
2.2.1 安装 JDK.....	23
2.2.2 安装 Eclipse.....	23
2.2.3 安装数据库客户端工具.....	24
2.2.4 安装 SSH 工具.....	24
2.2.5 安装其他工具.....	24
2.3 Linux 服务器环境搭建.....	25
2.3.1 安装 JDK.....	25
2.3.2 安装 Tomcat.....	26
2.3.3 安装 MySQL.....	26
2.3.4 安装 Mongo.....	28
2.3.5 安装 Redis.....	29
2.3.6 安装 Memcache.....	32
2.4 总结.....	33

## 入门篇 游戏开发

第3章 网络通信.....	36
3.1 通信协议.....	36
3.1.1 面向连接的 TCP.....	37
3.1.2 面向数据报的 UDP.....	38
3.1.3 HTTP 编程.....	39
3.1.4 Socket 编程.....	46
3.1.5 WebSocket 编程.....	54
3.2 Java NIO 基础.....	58
3.2.1 BIO 编程 (Blocking-IO, 阻塞式 IO).....	59
3.2.2 NIO 编程 (Non-Blocking IO, 非阻塞式 IO).....	61
3.2.3 AIO 编程 (Async IO/NIO.2, 异步 IO).....	68
3.3 Mina 的介绍及其使用.....	75
3.3.1 总体架构.....	76

3.3.2	IoService .....	77
3.3.3	IoFilterChain .....	77
3.3.4	IoHandler .....	77
3.3.5	IoSession .....	77
3.3.6	工作原理 .....	78
3.3.7	Acceptor 与 Connector 线程 .....	78
3.3.8	Processor 线程 .....	78
3.3.9	线程模型 .....	79
3.3.10	请求的处理顺序 .....	80
3.3.11	Mina 编程 .....	81
3.4	Netty 的介绍及其使用 .....	88
3.4.1	总体架构 .....	88
3.4.2	零拷贝 .....	89
3.4.3	Codec 框架 .....	90
3.4.4	Channel .....	90
3.4.5	ChannelEvent .....	91
3.4.6	ChannelPipeline .....	91
3.4.7	Netty 编程 .....	91
3.5	总结 .....	100
第 4 章	数据交互 .....	101
4.1	数据传输格式 .....	101
4.2	JSON 的使用及解析 .....	103
4.2.1	JSON 语法 .....	103
4.2.2	JSON 对象 .....	104
4.2.3	JSON 数组 .....	104
4.2.4	Java 中的 JSON 解析 .....	105
4.3	XML 的使用及解析 .....	110
4.3.1	XML 的特征 .....	111
4.3.2	数据共享 .....	111
4.3.3	数据传输 .....	111

4.3.4	平台兼容 .....	111
4.3.5	JSON 与 XML 的比较 .....	112
4.3.6	Java 中的 XML 解析 .....	112
4.4	Google Protocol Buffer 的介绍及使用 .....	128
4.4.1	Protobuffer 的安装与编译 .....	128
4.4.2	Protobuffer 的语法 .....	129
4.4.3	生成 Java 类 .....	130
4.4.4	Eclipse 的 protobuf-dt 插件 .....	131
4.4.5	示例程序 .....	132
4.5	总结 .....	134
第 5 章	数据缓存与持久化 .....	135
5.1	游戏数据存储 .....	135
5.1.1	数据分类 .....	136
5.1.2	数据缓存方式 .....	136
5.1.3	数据持久化方式 .....	137
5.1.4	数据库的比较 .....	137
5.2	MySQL 的介绍及使用 .....	138
5.2.1	特点 .....	138
5.2.2	数据类型 .....	139
5.2.3	MySQL 的使用 .....	139
5.2.4	在 Java 中使用 MySQL .....	142
5.3	MongoDB 的介绍及使用 .....	157
5.3.1	MongoDB 的主要特点 .....	157
5.3.2	了解 API .....	159
5.3.3	Mongo 的使用 .....	162
5.4	Memcache 的介绍及使用 .....	174
5.4.1	Memcache 的特点 .....	175
5.4.2	Memcache 的使用场景 .....	176
5.4.3	在 Java 中使用 Memcache .....	177
5.4.4	客户端使用要点 .....	182

5.5 Redis 的介绍及使用 .....	183
5.5.1 Redis 的特点 .....	183
5.5.2 Redis 的持久化 .....	184
5.5.3 Redis 的主从复制 .....	184
5.5.4 在 Java 中使用 Redis .....	185
5.6 总结 .....	199
<b>第 6 章 游戏逻辑 .....</b>	<b>200</b>
6.1 逻辑架构 .....	200
6.1.1 项目目录 .....	200
6.1.2 模块介绍 .....	202
6.2 逻辑流程 .....	212
6.2.1 网络模块 .....	212
6.2.2 线程池 .....	221
6.2.3 启动服务器 .....	222
6.2.4 逻辑请求处理 .....	223
6.2.5 关闭服务器 .....	228
6.3 事件处理器 .....	229
6.4 定时任务 .....	236
6.5 RPC 框架 .....	244
6.5.1 Json-rpc .....	244
6.5.2 Motan .....	253
6.6 总结 .....	264
<b>第 7 章 游戏安全 .....</b>	<b>265</b>
7.1 游戏安全的必要性 .....	265
7.2 登录安全 .....	266
7.3 游戏充值 .....	266
7.4 SQL 注入 .....	267
7.5 通信协议与消息格式 .....	268
7.6 整型溢出 .....	269

7.7 并发请求.....	269
7.8 逻辑漏洞.....	270
7.9 日志系统.....	271
7.10 总结.....	271

## 高级篇 游戏服务器的设计及优化

第8章 服务器架构分析.....	274
8.1 服务器架构的演变过程.....	274
8.2 全区同服架构分析.....	277
8.2.1 COC 架构模型分析.....	278
8.2.2 COK 架构模型分析.....	279
8.3 分区分服架构分析.....	281
8.4 弱联网类游戏架构分析.....	282
8.5 MMORPG 类游戏架构分析.....	283
8.6 总结.....	285
第9章 《皇室战争》游戏开发实战.....	286
9.1 微竞技游戏介绍.....	286
9.2 架构分析及搭建.....	287
9.2.1 功能分析.....	287
9.2.2 服务器部署架构.....	288
9.2.3 系统架构.....	289
9.3 数据持久化方案.....	290
9.3.1 数据结构分析.....	290
9.3.2 使用 Morphia 操作 MongoDB.....	295
9.4 Netty 网络框架的使用.....	300
9.4.1 Netty 实现的 HTTP 服务器.....	300
9.4.2 Netty 实现的 TCP 服务器.....	309
9.5 账号系统.....	316
9.6 个人信息.....	323

9.7 英雄卡牌系统.....	327
9.8 宝箱系统.....	334
9.9 战斗系统.....	339
9.10 客户端模拟.....	349
9.10.1 登录界面.....	349
9.10.2 选服界面.....	354
9.10.3 主逻辑界面.....	358
9.10.4 对战界面.....	363
9.11 总结.....	372
第 10 章 游戏开发技术前景 .....	373
10.1 Egret .....	373
10.2 Cocos 2D .....	374
10.3 Unity.....	375
10.4 Unreal.....	376
10.5 Java.....	376
10.6 Node.js.....	377
10.7 总结.....	378

# 基础篇

## 走进游戏开发

说到游戏，我相信大家都不陌生，有些人甚至因为小时候沉迷于游戏，而挨过父母的打骂。本章以认识游戏开始讲起，同时讲解了游戏行业现状及服务器的作用。



# 第1章

## 认识游戏

游戏是我们再熟悉不过的东西了，有的人每天都会打游戏，来打发无聊的时间，有的人会在紧张的工作之后玩玩游戏，缓解疲劳。但游戏究竟是一种什么样的东西呢？本章就来聊聊我们既熟悉又陌生的东西——游戏。

### 1.1

#### 什么是游戏

什么是游戏？这个问题似乎很简单，每个人都能说出一些答案，比如“游戏就是拿着手柄对着电视机狂按的小霸王游戏机”、“游戏就是放学后同学相约去网吧一起玩耍的东西”、“游戏就是大人不让小孩子玩的东西”。下面我就为大家介绍一下什么是游戏。

##### 1.1.1 游戏的定义

游戏并非只有电子游戏一种，跳皮筋、石头剪刀布等都属于游戏，只不过现在最流行的电子游戏被人们默认为游戏的代名词。实际上，任何娱乐活动，如打麻将、打牌等，都可以称为游戏。人类是高级灵长类动物，除了生活之外，还需要娱乐来丰富生活，于

是就产生了各种各样的游戏。所以我个人对游戏的定义就是：任何人类正常生理需求之外的活动均可称为游戏。下面我们来看看名人对于游戏的定义。

### 1. 柏拉图对游戏的定义

游戏是所有人或动物幼年时期增强生活能力需要所产生的有意识的模拟活动。

### 2. 亚里士多德对游戏的定义

游戏是劳作后的休息和消遣，是本身不带有任何目的性的一种行为活动。

### 3. 拉夫·科斯特对游戏的定义

游戏就是在快乐中学会某种本领的活动。

### 4. 辞海中关于游戏的定义

以直接获得快感为主要目的，且必须有主体参与互动的活动。

这个定义说明了游戏最基本的特性：

(1) 以直接获得快感（包括生理和心理的愉悦）为主要目的。

(2) 直接获得快感。我们能够从游戏中直接获取到让我们快乐的主观感受。

(3) 主体参与互动，指主体动作、语言、表情等变化与获得快感的刺激方式及刺激程度有直接联系。

游戏充斥着我们的生活，而且形式多样，但本书所讲的游戏只限于电子游戏。

## 1.1.2 游戏的分类

了解了游戏的定义之后，我们再来具体看看什么是电子游戏。电子游戏是时代发展的产物，它的产生大大丰富了人们的生活，使人们在闲暇时可以放松身心，这对生理及心理的调节都是很有帮助的，适当的游戏还能大大提高人们的工作效率（但过度沉迷于游戏就会损害人体的生理机能，如颈椎、眼睛等部位，这也是父母不让孩子玩游戏的重要原因）。随着时代变迁及硬件发展，电子游戏产品也从小霸王游戏机发展到 PC 端，再发展到现在的 PS2、手机、XBOX 等常见的电子游戏载体。

除了单机和网游的区别之外，游戏的类型也是多种多样的，下面就简单地对游戏类型做一介绍。

#### (1) RPG (Role-playing Game, 角色扮演游戏)

角色扮演类游戏是国内最流行的游戏类型，相信大家对这类游戏并不陌生，很多大型的网游都是角色扮演游戏。角色扮演能让玩家身临其境，有非常好的真实感。

举例：《剑灵》、《天下三》。

#### (2) ACT (Action Game, 动作游戏)

动作类游戏也是非常常见的，即玩家操控游戏中的角色进行各种激烈的打斗，在游戏中获胜之后会有一种很强烈的成就感。

举例：《拳皇》。

#### (3) AVG (Adventure Game, 冒险游戏)

由玩家控制游戏人物进行冒险，大多数冒险游戏以关卡形式体现，被玩家操控的游戏人物经历重重关卡阻碍，取得最终的胜利。

举例：《超级玛丽》、《魂斗罗》。

#### (4) FPS (First Personal Shooting Game, 第一人称视角射击游戏)

第一人称视角游戏是以玩家的视角进行射击类游戏操作。

第一人称视角射击能给人很强烈的即视感，玩过 FPS 游戏的人都能被它深深迷住。

举例：CS、《逆战》、CF、Left 4 Dead。

#### (5) TPS (Third Personal Shooting Game, 第三人称视角射击游戏)

第三人称视角射击是以玩家之外的视角来观察游戏人物，玩家能清晰地看到游戏人物的一举一动。

举例：《古墓丽影》。

#### (6) FTG (Fighting Game, 格斗游戏)

格斗游戏就是简单地让玩家操控游戏人物在游戏中相互格斗，最终打倒对方获得胜利。

举例：《拳皇》。

(7) SPT (Sports Game, 体育类游戏)

在 PC 上模拟各种体育活动的游戏。

举例：NBA2KOnline、FIFA。

(8) RAC (Racing Game, 竞速游戏, 也有人称其为 RCG)

在 PC 上模拟竞速, 比如摩托竞速、赛车竞速等。这类游戏通常能让玩家体验到现实生活中体验不到的速度感。

举例：《极品飞车》、《真实赛车》。

(9) RTS (Real-Time Strategy Game, 即时战略游戏)

这类游戏多是突出战术作用的 SLG 游戏的一个分支。

举例：《红警》、《星际争霸》。

(10) STG (Shooting Game, 射击类游戏)

这类游戏基本都是竖版的射击类游戏, 在手机上有更多的可玩性。

举例：《雷电》、《全民打飞机》。

(11) SLG (Simulation Game, 策略游戏)

这是目前手游中比较流行的一类游戏, 通常要求玩家有一定的水平, 在游戏中能展现更多的策略性, 对多操作的要求不是很高。

举例：COC、COK。

(12) MSC (Music Game, 音乐游戏)

这类游戏是为热爱音乐的人设计的, 玩家可在游戏中享受到美妙的音乐, 玩法也多种多样, 大部分游戏都是让玩家跟着音乐的节奏进行操作。

举例：《劲乐团》、《QQ 炫舞》、《节奏大师》、Deemo。

(13) SIM (Simulation Game, 生活模拟游戏)

与 SLG 不同, 此类游戏高度模拟现实, 能自由构建游戏中人与人之间的关系, 可以像现实生活一样进行人际交往, 还可以联网与众多玩家一起游戏。

举例：《模拟人生》。

#### (14) TCG (Trading Card Game, 育成游戏)

以前 GB 系列泛用，一般大家都用 EDU (Education) 来指代该类游戏，以便于和 TCG 区分开，这是玩家模拟培养的游戏。

举例：《美少女梦工厂》、《明星志愿》、《凌波丽育成计划》。

#### (15) CAG (Card Game, 卡片游戏)

玩家操纵角色通过卡片战斗模式进行的游戏。丰富的卡片种类使游戏更多变，给玩家带来无限的乐趣。

举例：《炉石传说》。

#### (16) LVG (Love Game, 恋爱游戏)

这是主要为男性制作的游戏。

举例：《心跳回忆》、《思君》、《青涩宝贝》、《秋忆》。

#### (17) GAL (Girl And Love GAME, 美少女游戏)

这类游戏盛产于日本，在我国几乎都是不符合法律要求的游戏。

举例：《真实的女朋友》。

#### (18) WAG (Wap Game, 手机游戏)

目前，手机游戏与 PC 基本是同等级别的。

举例：《天子》。

#### (19) MMORPG (Massively Multiplayer Online Role Playing Game, 大型多人在线角色扮演游戏)

目前，市面上常见的网游几乎都属于这类游戏。

举例：《剑灵》、《剑网三》。

#### (20) ARPG (Action Role-playing Game, 动作角色扮演类游戏)

这是动作类游戏和角色扮演类游戏的综合体，是主要以动作为主的 RPG 游戏。

举例：《功夫熊猫》。

### (21) ETC (etc. Game, 其他类游戏)

指玩家互动内容较少，或作品类型不明了的的游戏类型，通常不需要玩家太多的操作，基本相当于看剧情的游戏。

举例：Minecraft Story Mode、《行尸走肉》。

### (22) 动漫游戏

以同名动漫为原型制作的游戏，也有一些动漫游戏纯粹以动漫人物与类似动漫的情节为主板，但以同名动漫为原型改编的游戏占的比例多一些。

举例：《舰娘》。

### (23) MOBA (Multiplayer Online Battle Arena, 多人在线战术竞技游戏)

MOBA 也被称为 Action Real-Time Strategy (Action RTS, ARTS)，是视频游戏 (Video Game) ——即时战略游戏 (RTS) 的一个子类。目前这类游戏风靡全球，通常玩家需要购买装备，通过三四十分钟一局的形式开始两队的对抗，一般需要团队合作。

举例：《英雄联盟》、DOTA 2。

上述各类游戏的简写并不需要记住，但需要大概了解各个类型的游戏及其特色，以便在开发时更准确地搭建游戏架构。后续章节会对以上几种游戏架构进行简单分析介绍。

从古至今，游戏从打牌、下棋，发展到小霸王游戏机，再发展到 PC 游戏、手机游戏，从来没有间断过。人类已经离不开游戏了，游戏开发者有义务为玩家开发出更优秀、更用心的游戏。

## 1.2

## 游戏开发及分工

认识了游戏之后，我们就来了解一下游戏的开发。标准的软件开发大致需要经历以下几个阶段。

### 1. 可行性与计划研究阶段

首先需要拟定可行性研究报告，制定该软件的开发目标和总要求，并从多个角度进行一系列可行性分析，判断项目是否可行。通过之后就要制订项目开发计划，将开发过程中的经费预算、模块划分、开发进度、所需硬件和软件条件编制成文档。

### 2. 需求分析阶段

与客户沟通项目具体需求，此阶段的目的是让开发者和客户对项目的理解达成一致，使最终产品与最初需求偏差不大。

### 3. 设计阶段

确定需求之后就可以由架构师进行架构设计，之后就开始设计具体技术及实现。

### 4. 开发阶段

架构搭建完成之后，便可以由开发者进行开发，应根据项目组内的分工进行开发。

### 5. 测试阶段

开发完成之后，需要进行整体的集成测试、系统测试及验收测试。（实际上从需求分析阶段开始，测试工作就已经开始了）

### 6. 系统验收

客户验收系统。

以上是所有软件开发大致的过程，传统软件开发需要软件需求分析师、产品经理、项目经理、架构师、技术支持、运维、测试工程师和软件销售。各岗位各司其职，在各个阶段完成各自的任务。游戏开发流程与上述流程大致相同，一款游戏的诞生也要经历可行性研究、需求分析、设计、开发、测试和系统验收几个阶段。但我认为，因为游戏 CP 商（游戏内容提供者）的客户是全国乃至全世界的玩家，所以游戏产品的推广还需要有商务推广阶段和游戏运营阶段。而且商务推广和游戏运营是决定一款游戏成功与否的重要阶段，好的运营可以把一款普通的 IP 推广到排行榜靠前的位置，也可以把本就很优秀的 IP 运营得一塌糊涂。

游戏开发的流程大概分为以下几步。



## 1. 产品立项

产品立项一般是由公司高层经过市场调查发现某一类产品很有前途，打算做这一类产品，然后高层领导开会决定的。这些人可能包括美术总监、产品总监、游戏制作人、技术总监等。一致认同这个项目有一定可行性之后（通常立项前会有人做了充分的可行性分析，并提供可行性分析报告），就开始任命主策划（相当于项目经理），然后再分配技术、美术、策划、测试等工作人员。

## 2. 游戏设计

主策划组织新队员开会之后，这个项目就算真正开始了，这时是系统策划最忙碌的时候，他们开始依据游戏的定位来策划不同的功能架构，然后再交由其他策划去填补框架，框架成型一部分之后，技术人员就可以开始讨论技术上的架构。同时，测试人员也开始对需求文档进行仔细研究。

传统软件开发中，可能需求分析和产品设计是划分模块来做的，而在游戏开发中，模块的划分更详细。游戏的产品设计职位叫作策划，策划又分为以下几种。

（1）主策划：项目主要的策划，主要负责游戏整体的策划和项目管理，类似软件开发中的项目经理。

（2）系统策划：负责游戏规则设计，各种玩法设定。

（3）数值策划：负责游戏的平衡性，通过平衡战斗数值、基础数值等使玩家得到更好的体验。

（4）文案策划：负责文案设计，包括物品信息、装备名等的文字包装。

（5）关卡策划：负责关卡设计，包括怪物分布、敌人 AI、陷阱分布、关卡难度等。

（6）脚本策划：负责程序脚本编写，包括技能脚本、怪物 AI、任务脚本等。

策划完游戏的产品设计之后，将其整理成文档交给开发者，技术人员按照文档内容去实现。

## 3. 开发阶段

传统软件开发中技术人员可能还需要划分前端开发和后端开发。如果是 Windows 窗体软件，前端可能是 C++ 工程师、C# 工程师，后端可能是 C++ 工程师或 Java 工程师。如



果是 Web APP 开发, 前端可能是 HTML 5 工程师, 后端可能是 Java 工程师或 PHP 工程师。如果是 iOS APP/Android APP 开发, 前端可能是 iOS 开发工程师, 后端可能是 Java 工程师或 PHP 工程师。而在游戏开发中, 前端可能是 C++ 工程师、C# 工程师或 JavaScript 工程师, 后端可能是 Java 工程师或 C++ 工程师。

由于开发的软件应用的领域不同, 所以需要的人员也不同。即使同一个技术, 在不同领域也有不同的应用, 比如 C++ 窗体开发和 C++ 服务器开发涉及的技术就是不同的, Java Web 开发和 Java 在游戏服务器的开发也是不同的。

当然, 也不乏全栈工程师这样的人才, 通常他们精通前后端开发, 甚至包揽 DBA 和运维的工作。不过, 我更赞同分工明确的做法, 大家各司其职, 齐心协力做好一件事情。

需求设计文档完成之后, 应交由美术部门进行原画设计等美术工作, 确定游戏风格, 对游戏中的美术内容进行制作。技术人员要对技术工作进行定位, 确定用什么技术实现会比较好。技术又分为前后端技术, 前端技术研究使用什么样的前端游戏引擎, 或者定制使用自研引擎(目前大的游戏公司如网易、完美等都有自己的游戏引擎), 后端研究使用什么样的网络框架, 对技术框架的设计都是基于游戏的定位。

举例来说, MMORPG 和 SLG 无论是前端还是后端, 可能都属于两种架构, MMORPG 大多属于 3D 游戏, 所以前端可能会使用 Unreal、Unity 等 3D 游戏引擎。而 SLG 大多属于 2D 游戏, 所以使用 Cocos2dx 或 Egret 这种 2D 引擎偏多。如果游戏的需求是市面上的引擎都不能满足的, 那么就需要自己研发游戏引擎。

同样, 在后端开发中, MMORPG 的玩家基本上都是在一个大地图中奔跑, 并且能看到其他玩家不断移动, 所以需要所有视野范围内的玩家实时数据同步, 这就需要强联网长连接来实现。

而 SLG 大都属于单机玩法, 使用长连接反而显得臃肿、浪费带宽流量, 所以使用弱联网连接更符合需求。总之, 每个游戏需求都需要一个与之对应的技术架构, 一个好的架构是一款游戏的良好开端。

架构设计完成之后, 就是技术人员持续的模块开发阶段, 一般情况下, 这个阶段以一两周为一个周期, 大一点的游戏可能需要一两个月甚至更长时间。开发阶段, 策划必须超前程序的开发, 即在程序开发此版本的时候, 策划必须策划出下一版本内容, 同时

也要支持程序的开发。美术部分也是与程序开发同步进行的，通常需要比程序超前一些，这样才能在开发中放入对应的资源，不使开发工作受到阻碍。

#### 4. 测试阶段

这里的测试阶段并不是游戏测试阶段，不是玩家可以来玩游戏了，而是项目组的测试人员对游戏的功能性和稳定性进行测试。测试阶段是贯穿整个开发流程的，从文档产生到程序的模块测试、集成测试、系统测试等，测试分析游戏中各种正常情况和极端情况下的 bug，再通过工具（如 bugfree）或文档将其交给对应模块开发者，开发者根据 bug 描述去修复，这个阶段可以一直迭代，直到 bug 数在可控范围内。常见测试方法有黑盒测试、白盒测试、Monkey 测试等。

#### 5. Alpha 测试

Alpha 测试即项目组的内部测试，测试对象可以是项目组成员、老板或项目组人员的亲朋好友等。先将游戏在自己的朋友圈内进行测试，测试完成后反馈问题给开发者去解决。（小公司一般会省去这个阶段）

#### 6. Beta 测试

Beta 测试即我们常见的游戏内测，是在游戏正式上线前让小部分玩家进行删档或不删档测试，收集真实的游戏数据，这部分测试玩家在游戏正式上线后通常也会得到一定的福利。

#### 7. 上线推广阶段

如果公司觉得游戏产品的测试数据不错（通常判断次日留存、七日留存或付费率等数据），便可以交给商务部门进行大规模推广，比如手游就可以通过腾讯应用宝、iOS 应用商店等进行推广。

#### 8. 运营阶段

游戏上线之后就是游戏运营成员展现实力的时候了，他们可以成千上万地向游戏中导入用户，让更多的玩家熟知这款游戏，并且在玩家玩游戏的时候不断推出各种各样的活动，让玩家感觉这款游戏有很大的发展空间、有更多的可玩性，比如在圣诞节推出圣诞老人领取礼物、在万圣节推出打败怪物获得奖励等活动。真正好的运营，可以带领玩家逐步深入游戏。

## 1.3

## 游戏行业现状分析

了解了游戏和游戏开发，可能大家就会问，游戏行业的发展前景到底如何，是否有前途？答案是肯定的。随着科技不断进步，游戏的硬件载体越来越丰富多样，玩家的消费水平也普遍提高。换句话说，玩家在游戏中越来越有消费能力，游戏行业仍然是暴利行业。

曾经一夜暴红的 Flappy Bird《围住神经病猫》等游戏使开发者一夜暴富。至于这些游戏为什么会火，我只能说它符合当时人们的游戏口味。因此，我们只要找对了玩家的口味，就有了一个赚钱的机会。当然，做游戏不能只以赚钱为目的，我们的终极目标是做出玩家真正喜欢的游戏。

现在的游戏行业在很多方面都有探索的空间，比如以下几个方面。

### 1. 虚拟现实设备及内容

虚拟现实（Virtual Reality，VR）并不是新生的概念，早在 20 世纪 60 年代，它就已经被提出来了。随着科技进步，不少科幻影视作品里都出现了人们对于 VR 的遐想。在家里，你就能感受金字塔的宏伟、埃菲尔铁塔的壮丽、阿尔卑斯山的伟岸，或者来一场浪漫的太空之旅，又或者体验坐过山车的刺激。似乎只要拥有 VR，人们就能拥有整个世界。不管是 VR 视频、VR 教育，还是 VR 游戏，各个 VR 厂商都开始着手创造这些绝妙的 VR 遐想。根据各种统计数据显示，预测 2016 年将成为 VR 元年，这一年会有大量的 VR 作品产生。

目前 VR 硬件设备还没有达到一个统一的标准，无论是大厂商，如 HTC Vive 或者 Oculus，还是一些小的硬件厂商，都在促使 VR 向更好的方向发展。在众人的努力与共同期待下，相信 VR 会有不错的前景。

在 VR 视频、VR 教育和 VR 游戏等一系列 VR 产业中，最能让人感受到真实的必定是 VR 游戏。游戏是能让人与虚拟世界产生互动的，而这种互动更能让人产生愉悦感。或许人们不用再耗费时间和精力，就能在家里打一场羽毛球或踢一场足球，而在 VR 世界中，这些虚拟的感受就如同真实世界一样，就像真的打了一场羽毛球或真的踢了一场足球。VR 游戏还能让人感受到更多真实世界中感受不到的东西，试想一下，在 VR 世界

中玩《我的世界》（一款高自由度的沙盒生存游戏）、CS（一款FPS游戏）、《极品飞车》（一款竞速游戏）等游戏是什么感受。如果这些能带来强烈主观感受的、沉浸式的游戏真以VR的形式出现在我们身边，那么游戏又将是另外一种定义。

就目前形势来看，VR的时代即将到来，你们做好迎接它的准备好了吗？

## 2. 移动游戏操控设备

游戏设备经历了从小霸王游戏机到PC再到手机的变迁过程，游戏的操作方式也由手柄到键盘再到手机触屏，也有PSP这种掌机游戏机。但是目前，手机仍是大多数人玩游戏的主要设备，而手机的屏幕大小等一些限制使游戏的操作方式被限制在点击和移动等简单的操作上。2014年，新游互联推出了智能无线游戏手柄，完美地解决了手游在手机触屏操控中存在的问题，这款设备在24小时内就获得了百万筹金。移动游戏的外部操控设备有着不小的发展前景，一个好的移动游戏配上一个好的操作外设，能让玩家获得更多良好的游戏体验，未来这样的设备会越来越多且越来越受欢迎。

## 3. 互联网+游戏创业

有数据统计，目前游戏行业的从业人员已超过30万，开发团队超过5000个。然而，以手游市场为例，只需5~10人便能组成一个团队，两三个月便能开发出一款游戏产品，即便如此，手游的存活率仅为千分之一。动辄几十万元的游戏引擎使用费，大公司对资源与渠道的垄断，削减了中小开发团队的竞争力。

小团队经常以开发游戏的数量来取得优势，或许4个游戏中只有1个能存活下来，并且达到较好的数据。如果这款游戏在一个渠道平均每天挣2000元，10个渠道每天就赚2万元，每个月就赚60万元。或许这个数据有些偏高，但如果成功的游戏不止一款，而是有三四款或十几款，那么去除游戏联运、购买资源等成本费，每个月的利润也是一个不小的数字，并且小团队的人数并不多，如果把这些钱均分给每个人，也是一笔可观的数额。当然，以上的推算都有一个前提，就是有那么一款或几款成功的游戏，如果你的任意一款游戏都打动不了玩家，那么玩家也不会埋单。

2010年，成立于芬兰的一家小游戏公司——Super Cell，由6位合伙人创建，创业之初不过十来个人，挤在一间几十平方米的小办公室开发游戏。他们最初仅凭借Hay Day和Clash Of Clans两款游戏，就创造了手游界的奇迹，公司收入从每月70万美元增长到每日250万美元，并且这种情况一直保持至今。最近其推出的一款卡牌实时对战策略手

游 Clash Royale, 更是锦上添花, 使公司每月都有几个亿的收入, 超过了 PC 单机游戏的龙头——EA。Super Cell 是一家真正的做游戏的公司, 每款游戏都是精品, 其凭借最前沿的创意和最出色的表现, 成为当今手游界的领头羊。

#### 4. 电竞直播平台

随着竞技类游戏的兴起, 电竞行业开始火爆, 世界各地都在举办电竞比赛和活动。随之而来的自然就是电竞直播平台的火热, 从电竞游戏、电竞赛事、内容制作到电竞直播, 已经形成了一条完整的产业链, 同时与电竞直播相关的广告等也能带来不少收益。

#### 5. 原创 IP

从现在的手游不难看出, 越来越多的影视、动漫、文学方面的作品, 与游戏产品紧密结合。通过 IP 授权即可将其他产业的资源品牌引入到游戏中, 打开了游戏行业的新方向。原创 IP 授权也就成了游戏行业的另一个发展方向。在未来, 或许每一部电视剧、每一部电影, 在上映的时候都会同步推出一款同名游戏, 影视作品越火爆, 越能带动同名游戏的火爆。IP+影视+游戏的产业链在未来或许是游戏行业的多条出路之一。

#### 6. H5 游戏

HTML5 游戏也叫 H5 游戏。2015 年, 白鹭时代在北京国际会议中心举办了 HTML5 生态大会, 大会介绍了 Egret 新的一整套工作流, 为 HTML5 游戏的发展作出了巨大的贡献, 并且暴风墨镜 CEO 黄晓杰也宣布 Web VR 实验室成立, 这也就意味着, 未来 HTML5 游戏不仅可能闻名于手游界, 也可能被广泛使用在 VR 领域。

如今 HTML5 十分火爆, HTML5 游戏也变得炙手可热, HTML5 有着跨平台兼容及轻量的特点, 并且从《愚公移山》的作品数据来看, HTML5 游戏实现付费也不是不可能的。Cocos2d 的 js 版本和白鹭的 Egret 引擎基本都有一整套成型的开发流。然而, 个人认为, HTML5 游戏也并非无所不能, 未来的游戏是 Native 更吃香, 还是 Web 更有吸引力? 现在来说还为时过早。HTML5 游戏虽然有着很好的跨平台兼容性, 但是其用户体验比 Native 游戏稍微差一些, 就 Cocos2dx 使用 C++ 语言和 Cocos2d-js 使用 JavaScript 语言来看, 在游戏性能上 JavaScript 目前还不能与 C++ 媲美。我们不能断言谁一定能主宰游戏开发领域, 但不管是谁统领了游戏开发领域, 都会对游戏发展作出巨大的贡献, 都值得每一个开发者学习。



## 1.4

## 游戏服务器开发要点

上面说到了游戏行业的发展，那么游戏服务器在这个行业中处于怎样的地位呢？前面已经讲过，游戏开发分为前端开发和后端开发，后端开发就是服务器开发。当然，这里所说的后端开发，指的是网络游戏的开发，单机游戏自然也就不存在服务器，整个游戏的逻辑完全由客户端来承担。网络游戏的产生造就了游戏服务器开发工程师这个职业。本书以 Java 服务器开发为基础，那么如何做好 Java 游戏服务器开发呢？你需要做到以下几点。

(1) 你必须是一个热爱游戏、有一定游戏逻辑的开发者。完全不了解游戏、不明白游戏逻辑的人不适合做游戏服务器开发工作，做开发最重要的就是理解需求，只有真正玩过游戏、热爱游戏的人，才会更快地明白游戏的一些逻辑。

(2) 你需要了解网络知识，掌握一定的 Java 网络编程知识，了解 OSI 参考模型、套接字、长连接和短连接等网络基础知识。网络是网络游戏的核心（后续章节会介绍网络相关知识）。

(3) 你需要对 Java 并发编程有一定的了解，Java 作为服务器开发工具必不可少的要涉及多线程开发，多个玩家在线的时候，不可能用单线程来处理所有的请求，这样服务器响应会非常慢甚至会出现掉线的情况，用户体验就会非常不好。

(4) 你需要明白 Java 的一些设计模式，比如常见的单例模式、工厂模式、观察者模式等。好的设计模式能让后台程序轻松应对游戏需求的变更。对于那些有硬编码习惯的人，需求一变更他们就会特别头疼。另外，设计模式也是程序开发时特别容易让人理解的内容，如果你的代码不只是你一个人查看和使用的，最好使用一些设计模式，以让别人更容易看懂。

(5) 你需要掌握数据库存储、缓存等游戏数据保存的途径。游戏中的热数据需要灵活使用缓存，冷数据需要持久化到数据库，因此你必须掌握至少一个数据库的使用方法和至少一个内存数据库的使用方法。目前常用的数据库有 MySQL、Mongo，内存数据库有 Redis、Memcache、Ehcache 等。

(6) 你需要根据游戏逻辑提供相应的 GM 运营工具，可以写 Windows 窗体程序，也

可以写 Web 程序进行 GM 管理运营。总之，你需要从你的游戏逻辑中提供一些 GM 接口，接入到 GM 运营工具中。

(7) 你需要掌握系统运维知识，服务器的部署、架构搭建、负载均衡、日志管理、数据备份恢复、灾难处理等，你都要掌握。如果你不甘于只做游戏逻辑的后端开发，系统运维知识也是必不可少的，至少对于服务器主程来说，这些知识是必不可少的。

以上七点是我总结的做 Java 游戏服务器开发需必备的一些技能。我们知道，Java 更多的是被应用到 Web 开发中，那么这两者有什么异同点呢？下面简单说一下，以帮助 Web 开发者快速转到游戏开发中来。

(1) 从第三方支持来说，Web 后台有很多成熟的第三方框架，开发者不需要关心底层控制器跳转的实现，只需要一个或几个配置文件，就能完成核心控制器的部分，而开发者只需要关注 Web 自身的业务逻辑，将逻辑与框架融合即可。使用框架不仅简化了控制层代码，又很好地实现了业务逻辑的分层。而在游戏后台开发中，各种游戏的需求差异性很大，从网络层到业务逻辑层，各方面都必须根据游戏需求搭建适合的框架，因此很难有一些通用的内容能提炼出一款成熟的框架。游戏后台开发基本都需要自己搭建适合的框架，Web 项目的发布一般依赖 Tomcat 或 JBoss 这种 Web 容器，而游戏开发基本上都是自己实现一个 HTTP 服务器或 Socket 服务器。

(2) 从业务逻辑层面来说，Web 后台的逻辑都是大同小异的，或许这一套系统的逻辑稍微改一改，另一套系统就能用。而游戏就不同了，每个游戏都有自己的特色，要根据策划的不同需求实现不同的逻辑，虽然也会有一些通用模块，但整体的差异性还是很大的。

(3) 从数据持久化来说，Web 的数据基本上是很规整的，表与表之间关系很明确，并且以后也不会有太大的变化。而游戏中的数据多种多样，开服之后数据的变化也是多种多样的，传统的关系型数据库甚至根本无法满足游戏数据持久化的需求，游戏中有很多状态和数据是需要服务器来保存的。个人认为，在游戏开发中，NoSQL 比关系型数据库更实用。Web 开发更注重数据之间的对应关系，而对于游戏开发来说，数据库基本上只是玩家下线后数据保留的一个场所。

(4) 从通信层面来说，Web 中的用户都是一个个独立的个体，而游戏是多人在线的一个游戏世界。在这个游戏世界中，玩家与玩家之间需要进行交互，这就需要服务器实时地向所有在线玩家进行消息广播，这是非常占用带宽的，并且玩家的每一步操作都会

向服务器发起一个请求，这对服务器的性能要求也是非常高的。所以在这方面，游戏后台要比 Web 做更多的处理，游戏服务器大都属于 IO 密集型的。

## 1.5 总结

本书作为一本 Java 游戏服务器开发的书籍，先从游戏的产生讲起，然后讲解了游戏开发及分工，并对游戏行业的现状及未来发展进行了分析，最后介绍了游戏服务器开发在游戏开发中的作用，由浅入深地为大家开始 Java 游戏服务器开发做了一些简单的理论铺垫。下章我们将真正进入到开发实战环境，带领大家揭开 Java 游戏服务器开发的神秘面纱。



## 第2章

# 环境搭建

所谓“工欲善其事，必先利其器”，在深入学习 Java 服务器开发之前，首先要搭建好开发环境，并熟练掌握开发工具。本书讲解以 Eclipse 作为开发 IDE，使用 JDK1.6，并使用 Maven 管理项目，项目大体框架采用 Spring+Hibernate 进行搭建，其他部分框架依据具体情况使用。

### 2.1

## Windows 开发环境搭建

目前大部分开发者使用的都是 Windows 开发环境，下面先来讲解如何在 Windows 下进行环境搭建。

### 2.1.1 安装 JDK

我们要进行的必不可少的操作就是安装 JDK，并配上环境变量。JDK 的官方下载地址：<http://www.oracle.com/technetwork/java/javase/downloads/index.html>（本书所有环境都基于 JDK1.6 版本）。单击“jdk-6u5-windows-x64.exe”直接安装，安装时根据导航进行操作即可，本人不建议更换 java 默认安装目录。

安装完成后就开始配置环境变量，右击“我的电脑”，选择“属性—高级”选项，进行环境变量设置，我们将在这里设置 Java 的系统环境变量。

### 1. 设置 Path 变量

如果你想在任意目录下使用 Java 命令，就需要系统能随时找到 Java 的应用程序，在 Windows 环境下，就需要把这个应用程序的目录配置到环境变量中。如你的安装目录是 C:\jdk1.6.0，那么 Java 应用程序就在 C:\jdk1.6.0\bin 下，我们只需要把 C:\jdk1.6.0\bin 这个目录配置到环境变量的 Path 中，就能在任意目录使用 Javac、Java、Jawaw 等 Java 命令。

在系统变量里找到 Path 变量单击并编辑：

```
变量名: path  
变量值: C:\jdk1.6.0\bin;
```

### 2. 设置 CLASSPATH 环境变量

在最初的 JDK 版本中需要配置 CLASSPATH，是因为 JRE 还找不到它自带的类库，如 tools.jar 或 dt.jar，而我们在开发时又需要用到这些类库，所以需要配置 CLASSPATH。自 JDK1.5 版本之后，CLASSPATH 就不再是必须配置的变量，如果在 Eclipse 中开发，更不用配置 CLASSPATH，因为 Eclipse 自带我们需要的类库。这里我们还是配置一下，在环境变量中找到 CLASSPATH 遍历，如果没有就手动新建一个，并添加下列路径：

```
变量名: CLASSPATH  
变量值: .;%JAVA_HOME%\lib\tools.jar;%JAVA_HOME%\lib\dt.jar; (注意: 在 CLASSPATH 的最前面有一个 ".", 表示当前目录)
```

### 3. 设置 JAVA\_HOME

设置 JAVA\_HOME 可以方便设置其他目录时对 Java 目录的引用，如果设置 JAVA\_HOME 后要使用该路径，只需在变量中假设 %JAVA\_HOME% 即可，以避免重复输入路径字符串。一旦路径改变，只需改变 JAVA\_HOME 即可，如果没有 JAVA\_HOME，路径的改变可能就需要修改很多路径，一旦考虑不全，可能导致 Java 的环境配置错误。另外，很多第三方软件，如 Tomcat，会引用约定俗成的 JAVA\_HOME 作为 Java 的访问目录路径，如果没有 JAVA\_HOME，很可能需要自行修改其配置文件。

在系统环境变量一栏新建 JAVA\_HOME:

```
变量名: JAVA_HOME
变量值: C:\jdk1.6.0
```

通过以上步骤，就可以用 Windows 下的命令行来测试 JDK 环境是否安装好，按快捷键“Win+R”打开运行窗口，在运行窗口输入“cmd”，按回车键打开 Windows 的 Dos 命令终端窗口。输入命令 `java -version`，可以看到当前 JDK 的版本，如果提示“java 不是内部或外部命令，也不是可运行的程序或批处理文件”，那很可能是环境变量没有配置好。输入 `java` 命令，可以查看 `java` 命令的用法介绍；输入 `javac` 命令可以编译 `java` 源文件。如果以上命令都没有出现问题，那么基本上 JDK 环境就搭建好了。

## 2.1.2 安装 Eclipse

搭建好 JDK 开发环境之后，就可以开发 Java 程序了，我们可以用记事本写一个以 `.java` 结尾的 Java 源文件，并在命令行中用 `javac` 进行编译，然后用 `java` 命令运行编译好的 `.class` 文件。

如果将以上步骤作为项目开发流程，未免有些太烦琐了，并且也会非常不方便，这时，就要用到 IDE（Integration Development Environment，集成开发环境）。Java 最常用的 IDE 就是 Eclipse、NetBeans、IntelliJ IDEA 等。本书使用 Eclipse 作为 IDE，Eclipse 官方下载地址为 <http://www.eclipse.org/downloads/>。

Eclipse 是一个集成开发环境，它本身是基于 Java 的，并且是一个完全开放源代码的可扩展平台。Eclipse 本身只是框架或者说是一组服务，它是通过插件组件来构建的开发环境。Eclipse 自带一个标准的插件集，做 C++ 开发可以用 CDT 插件、做 Android 开发可以用 ADT 插件、做 Go 语言开发可以用 `goclipse` 插件、做 Node 开发可以用 `noclipse` 插件。Eclipse 的插件库已经比较完善，即使 Eclipse 在其他方面仍然存在很多缺陷，但它仍是非常受欢迎的一款完善的工作流工具。

Eclipse 还有很多快捷键，使用这些快捷键，能够提高工作效率，常用快捷键如表 2-1 所示。

表 2-1 Eclipse 常用快捷键

快 捷 键	说 明
Ctrl+I	快速修复

续表

快 捷 键	说 明
Ctrl+Shift+O	自动导入所需要的包
Ctrl+D	删除当前行
Ctrl+Alt+↓	复制当前行到下一行（复制增加）
Ctrl+Alt+↑	复制当前行到上一行（复制增加）
Alt+↓	当前行和下一行交互位置
Alt+↑	当前行和上一行交互位置
Alt+←	前一个编辑的页面
Alt+→	下一个编辑的页面
Shift+Enter	在当前行的下一行插入空行
Shift+Ctrl+Enter	在当前行插入空行
Ctrl+O	快速显示 OutLine
Ctrl+Q	定位到最后编辑的地方
Ctrl+/	注释当前行，再次使用此快捷键则取消注释
Ctrl+/（小键盘）	折叠当前类中的所有代码
Ctrl+×（小键盘）	展开当前类中的所有代码
Ctrl+Shift+X	把当前选中的文本全部变为小写
Ctrl+Shift+Y	把当前选中的文本全部变为大写
Ctrl+Shift+F	格式化当前代码

表 2-1 只整理了一小部分比较常用的快捷键，更多实用的快捷键需要在平时操作的时候记忆，熟练使用快捷键能很好地提升工作效率。

### 2.1.3 安装数据库客户端工具

利用 Java 进行服务器开发，必不可少地要用到数据库，利用数据库的一些客户端工具能提升开发效率。以下是几款常用的数据库客户端工具：

(1) Navicat for MySQL (MySQL)： <http://www.navicat.com.cn/>。

(2) MongoVUE (Mongo)： <http://www.mongovue.com>。

(3) RedisClient (Redis)： <https://github.com/caoxinyu/RedisClient>。

### 2.1.4 安装 SSH 工具

在服务器开发中需要使用 SSH 工具部署项目到开发服务器或正式服务器，SSH 工具

也有很多种，下面是两款 Windows 下的 SSH 工具（使用哪种工具全凭个人喜好）。

(1) XShell: [http://www.netsarang.com/products/xsh\\_overview.html](http://www.netsarang.com/products/xsh_overview.html)。

(2) SSH: [http://ultra.pr.erau.edu/~jaffem/tutorial/SSH\\_secure\\_shell\\_client.htm](http://ultra.pr.erau.edu/~jaffem/tutorial/SSH_secure_shell_client.htm)。

### 2.1.5 安装其他工具

在服务器开发中需要用到的或者能提高开发效率的工具都需要安装。

#### 1. 文本编辑器

Editplus: <https://www.editplus.com/download.html>。

Sublime Text: <http://www.sublimetext.com/>。

#### 2. 容器

Tomcat: <https://tomcat.apache.org/>。

JBoss: <http://jbossas.jboss.org/downloads/>。

#### 3. JSON 解析工具

在线 JSON 解析: <http://json.cn>。

## 2.2

### Mac OS X 开发环境搭建

OS X 是苹果公司的 Mac 系列产品的操作系统，这款操作系统只能运行在 Mac 系列产品的硬件之上。OS X 操作系统的宗旨是简洁。

1985 年，史蒂夫·乔布斯（Steve Jobs）被迫离开苹果公司后，成立了 NeXT 公司，并开发了面向对象操作系统——Openstep，后来苹果公司收购了 NeXT 公司，史蒂夫·乔布斯再次担任苹果公司 CEO，并将 Mac OS 与 Openstep 进行整合，开发出了全世界第一个基于 FreeBSD 的面向对象操作系统——OS X。

不管 Mac OS X 历史如何，个人认为，这个操作系统就是为程序员设计的，它有类似 Windows 的 GUI 界面，也有 UNIX 的性能，是兼容二者优点于一身的操作系统，可

以在 Mac OS X 上直接使用 UNIX 命令操作，并且在系统安装完成后，它就默认安装了 python、jdk、apache 等常用的开发环境，为开发者提供了很多便利。下面就开始学习如何在 Mac OS X 上搭建开发环境。

### 2.2.1 安装 JDK

上面说到 Mac OS X 安装完成后，就默认安装了 JDK 环境，但是我们还是有必要说明一下 JDK 的安装步骤。因为或许系统默认安装的 JDK 版本并不是我们想要的，此时就需要安装其他版本的 JDK。

Oracle 官网从 JDK1.7 才有了 Mac 版本的安装包，但有的项目却要使用 JDK1.6，此时就可以通过其他方法安装 JDK1.6。Apple 的开发者网站提供了 JDK1.6 的安装包，可以在此下载 JDK1.6。

下载地址：<http://connect.apple.com/>。

默认的 JRE 路径：`/System/Library/Frameworks/JavaVM.framework/Versions/CurrentJDK/Home`。

Apple 的 JDK 路径：`/Library/Java/JavaVirtualMachines/`。

JAVA\_HOME 路径：`/Library/Java/JavaVirtualMachines/1.6_29.jdk/Contents/Home`。

与 Windows 系统一样，在 Mac OS X 系统下，仍然需要配置 JAVA\_HOME 到系统环境变量。在 UNIX 系列中，操作系统最重要的文件就是 `/etc/profile`，这个文件包含一些重要的配置信息，需要在这个文件中添加以下代码（以 1.6\_29 开头的文件夹名字代表你的 Java 版本）：

```
JAVA_HOME=/Library/Java/JavaVirtualMachines/1.6_29.jdk/Contents/Home/  
export JAVA_HOME
```

### 2.2.2 安装 Eclipse

做 Java 开发必不可少的还是 Eclipse，Eclipse 是基于 Java 编写的 IDE，也有 Mac 版本的。在 Eclipse 官网就有 Mac 版本的下载链接：<http://www.eclipse.org/downloads/>。



其他内容，如 Eclipse 快捷键等，可以参照 Windows 环境搭建中的 Eclipse 安装部分。

### 2.2.3 安装数据库客户端工具

在 Mac 端也有很多数据库客户端工具。

- (1) Navicat for MySQL for Mac (MySQL) : <http://www.navicat.com/download>。
- (2) Redis Desktop Manager (Redis) : <http://redisdesktop.com/>。
- (3) RockMongo (Mongo) : <http://rockmongo.com/>。

### 2.2.4 安装 SSH 工具

Mac 虽然没有 XShell，但其有很好用的 Zoc。

Zoc: <http://www.emtec.com/download.html#zocfiles>。

### 2.2.5 安装其他工具

#### 1. 文本编辑器

Textmate: <http://macromates.com/>。

Sublime Text: <http://www.sublimetext.com/>。

Mac 自带的 Vim 也是很好用的。

#### 2. 容器

Tomcat: <https://tomcat.apache.org/>。

JBoss: <http://jbossas.jboss.org/downloads/>。

#### 3. JSON 解析工具

在线 JSON 解析: <http://json.cn>。

在 Mac 下搭建开发环境与 Windows 环境差不多，但由于存在一些系统环境的差异，有些内容的安装可能稍微复杂，不过熟悉 UNIX 系统的读者应该很快就能安装好。

## 2.3 Linux 服务器环境搭建

以上两节内容均为开发环境的搭建，但服务器的开发除了要求在本机运行没有问题之外，还要求部署项目到服务器上时也能平稳运行。所以，我们除了在本机搭建开发环境之外，还需要在服务器搭建项目的运行环境。本节主要讲解常用的环境及数据库等的安装与配置。

### 2.3.1 安装 JDK

Java 开发必不可少的是 JDK (Java Development Kit)，Linux 系统下的 JDK 安装与 Mac OS X 下的 JDK 安装步骤大致相同，首先需要在官网下载需要的 JDK 版本。

官方下载地址：<http://www.oracle.com/technetwork/java/javase/downloads/index.html>。

官方的 JDK 安装包有多种版本，这里以后缀为 bin 的 JDK 安装包为例：

(1) 在 Linux 服务器上创建 JDK 的文件夹：mkdir /usr/java，然后将下载好的 bin 文件复制到文件夹中。

(2) 更改文件权限：chmod 755 jdk-6u11-linux-i586.bin。

(3) 执行安装包文件：./jdk-6u11-linux-i586.bin。

(4) 执行过程中，需要按照提示输入“yes”或“y”进行解压。

(5) 安装完成后，JDK 会被安装到/usr/java 文件夹中，然后需要配置系统环境变量。打开/etc/profile 文件，配置 Java 的环境变量：

```
#打开 /etc/profile，在里面添加如下内容
export JAVA_HOME=/usr/java/jdk1.6.0_27
export JAVA_BIN=/usr/java/jdk1.6.0_27/bin
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

```
export JAVA_HOME JAVA_BIN PATH CLASSPATH
让/etc/profile 文件修改后立即生效，可以使用如下命令：
# . /etc/profile
```

输入命令 `java -version`、`javaw`、`javac` 即可检验 JDK 环境是否安装完成。

## 2.3.2 安装 Tomcat

如果 Java 程序都有 Web 服务器，就需要 Tomcat 这样的容器进行加载，Tomcat 的 Linux 版本也可在其官网上下载。

官方下载地址：<http://tomcat.apache.org/download-60.cgi>。

其也有很多版本，下面以 tar.gz 版本为例进行讲解。（选择 Binary Distributions 下的文件）

（1）在 Linux 服务器中创建目录 `mkdir /usr/tomcat`，并将安装包复制进去。

（2）解压安装包到目录：`tar -zxvf apache-tomcat-6.tar.gz`，解压完成后，进入解压出来的目录。

`conf` 目录是 Tomcat 的配置文件目录，`server.xml` 就是 Tomcat 的核心配置文件，通过 `server.xml` 的 `Connector` 可以配置端口号、最大线程、最大等待时间等参数。

`Webapp` 目录是 Tomcat 发布项目的目录。

`logs` 目录是 Tomcat 运行时的日志记录。

`bin` 目录是 Tomcat 核心脚本文件目录，`startup.sh` 是它的启动脚本，`shutdown.sh` 是它的关闭脚本。

关闭 Tomcat 的配置和优化的内容还有很多，这里不再赘述，只要配置好 `Connector` 端口号即可，其他性能优化可参照网络上的资料。

## 2.3.3 安装 MySQL

数据库也是服务器必不可少的环境之一，这里先介绍 MySQL 的安装与配置。

## 1. 下载编译安装

官方下载地址: <http://dev.mysql.com/downloads/>。

```
#cd /usr/local/src/
#wget http://mysql.byungsoo.net/Downloads/MySQL-5.1/mysql-5.1.38.tar.gz
#tar -xzf mysql-5.1.38.tar.gz ../software/
#./configure
--prefix=/usr/local/mysql //MySQL 安装目录
--datadir=/mydata //数据库存放目录
--with-charset=utf8 //使用 UTF8 格式
--with-extra-charsets=complex //安装所有的扩展字符集
--enable-thread-safe-client //启用客户端安全线程
--with-big-tables //启用大表
--with-ssl //使用 SSL 加密
--with-embedded-server //编译成 embedded MySQL library (libmysqld.a)
--enable-local-infile //允许从本地导入数据
--enable-asm //汇编 x86 的普通操作符, 可以提高性能
--with-plugins=innobase //数据库插件
--with-plugins=partition //分表功能, 将一个大表分割成多个小表
#make && make install //编译, 然后安装
```

## 2. 启动 MySQL

```
#/usr/local/mysql/bin/mysqld_safe --user=mysql & //启动 MySQL
```

## 3. 配置开机启动

```
#cp /usr/local/src/software/mysql-5.1.38/support-files/mysql.server /etc/
init.d/mysqld
#chmod 755 /etc/init.d/mysqld
#chkconfig --add mysqld
#chkconfig mysqld on
#service mysqld restart
```

## 4. 登录测试

```
#cd /usr/local/mysql/bin
#mysql
>show databases;
```

### 5. 设置初始密码

6. MySQL 安装完成后, 需要对 root 账户设置初始密码, 无密码登录后执行下面的 sql 语句

```
mysql> set password for root@localhost=password('newpassword');
```

至此, MySQL 的安装就基本完成了。

## 2.3.4 安装 Mongo

目前热度最高的 NoSQL 数据库——Mongo 的安装步骤如下。

### 1. 下载安装包

下载安装包:

```
wget http://fastdl.mongodb.org/linux/mongodb-linux-i686-1.8.2.tgz
```

下载完成后解压压缩包:

```
tar xzf mongodb-linux-i686-1.8.2.tgz
```

### 2. 安装准备

将 mongodb 移动到/usr/local/server/mongodb 文件夹:

```
mv mongodb-linux-i686-1.8.2 /usr/local/mongodb
```

创建数据库文件夹与日志文件:

```
mkdir /usr/local/mongodb/data  
touch /usr/local/mongodb/logs
```

### 3. 设置开机自启动

将 mongodb 启动项目追加加入 rc.local, 保证 mongodb 在服务器开机时启动:

```
echo "/usr/local/server/mongodb/bin/mongod --dbpath=/usr/local/server/  
mongodb/data -logpath=/usr/local/server/mongodb/logs -logappend --auth  
-port=27017" >> /etc/rc.local
```



#### 4. 启动 Mongo

进入 `mongodb` 目录下的 `bin` 文件夹，然后启动 `mongodb`。

```
//下面是需要权限的登录方式，用户连接需要用户名和密码
/usr/local/server/mongodb/bin/mongod
--dbpath=/usr/local/server/mongodb/data
--logpath=/usr/local/server/mongodb/logs
--logappend --auth --port=27017 --fork
//下面是不需要密码的登录方式
/usr/local/server/mongodb/bin/mongod
--dbpath=/usr/local/server/mongodb/data
--logpath=/usr/local/server/mongodb/logs --logappend --port=27017 --fork
```

#### 5. 登录测试

进入 `mongodb` 目录下的 `bin` 文件夹，然后执行命令 `./mongo`。

运行如下：

```
[root@namenode mongodb]# ./bin/mongo
MongoDB shell version: 1.8.2
connecting to: test
> use test;
switched to db test
```

### 2.3.5 安装 Redis

服务器中还经常用到内存数据库——Redis，其安装配置步骤大致如下。

#### 1. 下载 Redis

下载地址：<http://code.google.com/p/redis/downloads/list>。

#### 2. 安装 Redis

下载后解压 `tar zxvf redis-1.2.6.tar.gz` 到任意目录，例如 `/usr/local/redis-1.2.6`

解压后进入 Redis 目录：

```
cd /usr/local/redis-1.2.6 make
```



### 3. 复制配置文件

在 Redis 的目录下有一个 redis.conf 配置文件，这是 Redis 给出的 Redis 默认配置的示例，我们可以把这个文件复制到 etc 下。

```
cp redis.conf /etc/
#把需要用到的 redis 程序复制到系统环境，和 Windows 下的环境变量配置相似，也是为了能在任意目录使用 Redis 命令
cp /redis-cli redis-server /usr/bin/
```

### 4. 改变 Redis 的内存分配策略

编辑文件 /proc/sys/vm/overcommit\_memory。（此文件存储 Redis 的内存分配策略）

可选值：0、1、2。

其中，

0——内核会先检查可用内存是否足够，如果可用内存已经足够，那么内存申请会被允许；如果可用内存不够，则内存申请会失败，并返回错误信息到应用进程。

1——无论当前的内存状态如何，内核都允许给 Redis 分配所有的物理内存。

2——内核分配的内存大小超过物理内存和交换空间总和。

Redis 在 dump 数据的时候，会先 fork 一个子进程出来。理论上，这个子进程占用的内存与父进程一样。如果内存负担不了，往往就会造成 Redis 服务器宕机或者 IO 超负载，使 Redis 服务器的工作效率下降，因此在设置内存分配策略的时候，我们通常会选择 1 的方式。

5. 开启 Redis 端口，修改防火墙配置文件（云服务器还需要修改后台管理中的防火墙限制）

```
vi /etc/sysconfig/iptables
```

加入端口配置：

```
-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 6379 -j ACCEPT
```

重新加载规则（重启 iptables 服务）：

```
service iptables restart
```

## 6. 启动 Redis 服务

```
[root@Architect redis-1.2.6]# pwd
/usr/local/redis-1.2.6
[root@Architect redis-1.2.6]# redis-server /etc/redis.conf
```

查看进程，确认 Redis 已经启动：

```
[root@Architect redis-1.2.6]# ps -ef | grep redis
Root    40129222      0 18:06 pts/3    00:00:00 grep redis
Root    29258         1 0 16:23 ?        00:00:00 redis-server /etc/redis.conf
```

如果 `redis.conf` 文件有问题，会造成 Redis 服务器启动失败，通常会找一个可用的配置文件直接覆盖，以免遇到很多坑。建议通过 `daemon=true` 将 `redis.conf` 文件设置为后台守护进程。

## 7. 测试 Redis

```
[root@Architect redis-1.2.6]# redis-cli
redis> set name hejincheng
OK
redis> get name
"songbin"
```

## 8. 关闭 Redis 服务

```
redis-cli shutdown
```

Redis 服务关闭后，内存中缓存的数据就会自动 dump 到硬盘上，在 `redis.conf` 中，可以配置 `dbfilename dump.rdb` 来改变缓存数据 dump 到硬盘中的具体目录。如果需要强制备份数据到磁盘，可以使用如下命令：

```
redis-cli save
```

或者，

```
redis-cli -p 6380 save (指定端口)
```

Redis 的具体配置还要根据服务器需求来决定，后面章节会具体讲解 Redis 的相关配置选择。

## 2.3.6 安装 Memcache

高效的内存数据库 Memcache 也是服务器常用的组件之一，它的安装配置步骤如下。

### 1. 安装 libevent

官方下载地址：<http://monkey.org/~provos/libevent/>。

Memcache 的运行需要依赖 libevent，因此先下载安装 libevent。

创建 libevent 目录：`mkdir /home/libevent`，复制下载好的压缩包到目录中。

```
#解压 tar zxvf libevent-1.4.14b-stable.tar.gz
#安装 libevent
cd libevent-1.4.14b-stable
./configure --prefix=/home/libevent/libevent-1.4.14b-stable
make
make install
```

执行完以上命令之后，如果一切顺利，就证明 libevent 安装成功。

### 2. 安装 Memcache

官方下载地址：<http://memcached.org/>。

创建 Memcache 目录：`mkdir /home/memcache`，复制下载好的压缩包到目录中。

```
#解压 tar zxvf memcached-1.4.5.tar.gz
#安装 memcache
cd /home/memcache/memcached-1.4.5
./configure --prefix=/home/blue/memcached-1.4.5 --with-libevent=/home/blue/
libevent-1.4.14b
make
make install
```

### 3. 启动 Memcache 服务

进入 bin 目录，执行 `./memcached -d -m 1024 -u blue`。

以下是 memcached 命令的参数：

```
# /usr/local/bin/memcached -d -m 200 -u root -l 192.168.1.91 -p 12301 -c
1000 -P /tmp/memcached.pid
```

相关解释如下：

- d——启动一个守护进程。
- m——分配给 Memcache 使用的内存数量，单位是 MB。
- u——运行 Memcache 的用户，如果当前为 root，需要使用此参数指定用户。
- l——监听的服务器 IP 地址。
- p——设置 Memcache 监听的端口。
- c——最大运行的并发连接数，默认是 1024。
- P——设置保存 Memcache 的 pid 文件。
- d install——安装 memcached。
- d uninstall——卸载 memcached。
- d start——启动 memcached 服务。
- d restart——重启 memcached 服务。
- d stop——停止 memcached 服务。
- d shutdown——关闭 memcached 服务并终结进程。

## 2.4 总结

通过本章内容的学习，我们大致能够搭建起一个 Windows 或 Mac OS X 的开发环境和一个 Linux 下的服务器部署环境。在实际项目中可能还会用到其他的软件或服务，可根据实际情况搭建环境。下面我们将正式开始游戏服务器开发中网络层部分的学习。

# 入门篇

## 游戏开发

接下来将分模块对 Java 服务器开发进行介绍, Java 服务器按照模块可以划分为网络层、传输层、数据层、逻辑层和安全层, 并通过实例代码分别进行介绍。学习本章可以了解到 Java 服务器开发中常见的模块开发内容。

# 第3章

## 网络通信

网络是网络游戏服务器最核心的模块，没有网络就没有网络游戏，网络游戏主要是多人在线对抗或合作游戏，因此要时刻保持网络通畅。本章分别从通信协议、通信模型、Java Nio 基础和网络框架等方面对网络层的开发进行介绍。

### 3.1

#### 通信协议

在网络游戏中，游戏客户端与游戏服务器需要进行通信，而通信就需要通信协议。这就好比战争时代的通信密报，需要遵循双方的规定对要发送的内容进行编码，对要接收的内容进行解码。而在网络游戏的开发中，客户端与服务器端通过一个共同的通信协议收发数据，以保证双方进行正常的交流。交流什么、怎样交流及何时交流，都必须遵循某种能接受的规则，即通信协议。

国际标准化组织（ISO）制定了 OSI（Open System Interconnection）网络模型，将网络通信分为 7 层，分别是物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。该模型定义了不同计算机互联的标准，是设计和描述计算机网络通信的基本框架。



OSI 模型将网络通信进行了详细划分，但在实际应用中却只用到了 4 层，于是就出现了 TCP/IP 网络模型。TCP/IP 网络模型是互联网的基础，是一系列网络协议的总称。TCP/IP 模型的层次如下。

### 1. 链接层

对应于 OSI 参考模型中的物理层和数据链路层。它主要负责监视数据在主机和网络之间的交换。该层主要有地址解析协议（ARP）。

### 2. 网络层

对应于 OSI 参考模型的网络层。它主要解决主机和主机之间的通信问题。它所包含的协议设计数据包负责在整个网络上的逻辑传输。它还负责数据包在多种网络之间的路由。该层有三个主要协议：网际协议（IP）、互联网组管理协议（IGMP）和互联网控制报文协议（ICMP）。

### 3. 传输层

对应于 OSI 参考模型的传输层。它为应用层实体提供端到端的通信功能，保证了数据包的顺序传送及数据的完整性。该层定义了两个主要协议：传输控制协议（TCP）和用户数据报协议（UDP）。

### 4. 应用层

对应于 OSI 参考模型的会话层、表示层和应用层。它提供用户需要的各种服务，是与用户最近的协议层，如 FTP、Telnet、DNS、SMTP、HTTP、Socket 等。

正是这些通信协议，才保证了端对端网络通信的流畅。我们经常使用的是 TCP 和 UDP，它们是网络应用中常见的协议，都属于传输层，但二者却有很大的差异。

## 3.1.1 面向连接的 TCP

TCP 是一种基于连接的协议，也就是说，在双方通信传输数据之前，必须要建立起可靠的连接。例如，在双方进行通话之前，必须保证电话是接通的状态。主机 A 和 B 如果要建立 TCP 连接，需要进行三次对话，大致过程如图 3-1 所示。

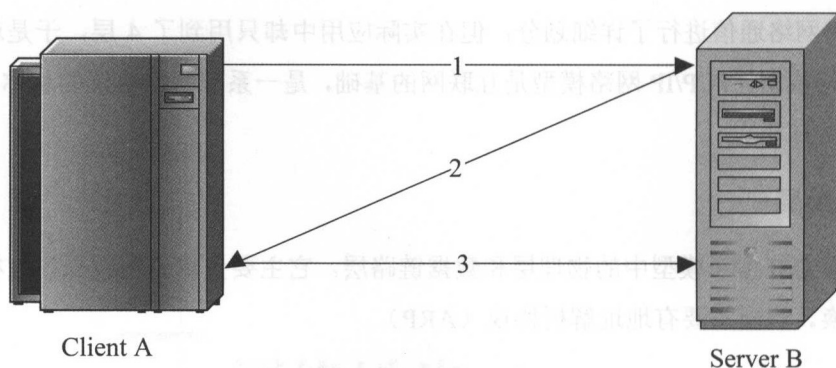


图 3-1 TCP 握手过程

(1) A 询问 B，发送请求建立连接数据包。

(2) B 应答 A，返回同意或拒绝建立连接数据包。

(3) B 如果同意，A 就发送建立连接数据包。B 如果拒绝，会话结束。

经过以上三步建立连接之后，便建立起一条隐形的通信通道，双方的连接便不会断开，双方都可以收发数据。

TCP 主要有以下几个特点。

(1) 面向连接：通信之前必须通过三次“握手”建立可靠连接。

(2) 安全可靠：每一次通信都必须得到对方的应答，否则认为数据报丢失，需要重发。

(3) 全双工通信：一旦建立连接，双方都可以通过通道进行数据传输。

(4) 一对一：通信只能建立在两个点之间。

(5) 面向流通信：通信传输是通过流的形式进行的。

### 3.1.2 面向数据报的 UDP

UDP 无须建立连接，只要指定目标地址，即可通过 UDP 向目标地址发送数据报。由于没有建立可靠连接，不保证数据报可以被送到目的地，因此数据是有可能丢失的。但和 TCP 相比，UDP 可以发送更大的数据报，并且可以进行一对多的广播发送。

UDP 主要有以下几个特点。

(1) 无连接：通信之前无须建立可靠连接。

(2) 数据无保障：UDP 不对数据进行排序，UDP 报文头部中无报文顺序相关信息，并且报文不一定按顺序到达，因此可能造成报文混乱。

(3) 开销小：由于无连接，并且不保证报文送达和报文顺序等，因此 UDP 的开销更小，比 TCP 的传输速度更快。

(4) 一对一，一对多，多对多：由于 UDP 无须建立连接，因此可以进行一对一通信，也可以进行一对多的广播和多对多的通信。

和 TCP 相比，虽然 UDP 看似有很多缺陷，但是在不同的应用场景中，TCP 和 UDP 各有优势，大部分情况下，如登录、支付、上传等，都需要服务器返回具体的执行结果以判断操作是否成功，这就需要使用 TCP。但在电视直播中，如果每一帧或每几帧画面都需要在直播服务器发送之后得到确认，不仅会使画面造成卡顿，对网络带宽的占用也是一种资源浪费。因此，我们要根据具体的应用场景来判断使用何种通信协议。

开发者除了要了解常见协议的理论之外，最主要的是理解并灵活使用应用层的协议，Java JDK 提供了使用这些协议的 API，根据这些 API 就能开发出基于这些协议的网络程序。下面对三种常见的应用层协议进行介绍。

### 3.1.3 HTTP 编程

互联网上应用最广泛的一种网络协议——HTTP，是最常见的超文本传输协议，几乎所有的 WWW 文件都必须遵守这个标准。设计 HTTP 最初只是为了提供一种 HTML 页面发布和接收的方法。HTTP 的工作原理很简单，客户端请求服务器建立连接并发送数据，服务器接收到请求并进行处理，处理完成之后返回数据并断开连接。下面以访问网站的流程来举例说明（见图 3-2）。

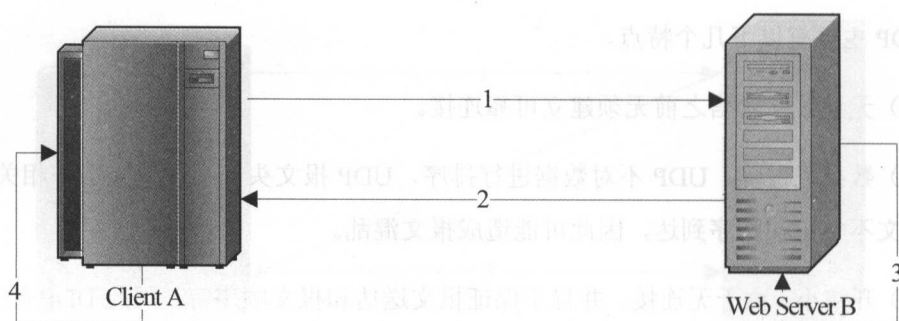


图 3-2 HTTP 浏览网页过程

(1) A 在浏览器输入网址，按回车键，浏览器封装 URL 到 HTTP 请求体并请求 B。

(2) B 接收到请求后建立连接，将请求中 head 及 body 体中的信息经过服务器逻辑处理后，返回 HTML 页面给 A。

(3) A 接收到 B 返回的 HTML 页面，断开连接。

(4) A 通过浏览器解析呈现在浏览器中。

#### 【范例】

```

package com.hjc.http;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;

/**
 * java.net 包中是与网络操作相关的 API
 *
 */
public class MainClass {
    public static void main(String[] args) {
        try {
            String destination = "http://www.baidu.com";
            // 定义 Url
            URL url = new URL(destination);
            // 打开连接，强制转换为 HttpURLConnection

```

```

URLConnection conn = (URLConnection) url.openConnection();
conn.connect();
// 获取输入流, 并用 BufferedReader 进行封装
BufferedReader reader = new BufferedReader(new InputStreamReader(
    conn.getInputStream()));
StringBuffer sBuffer = new StringBuffer();
String line = "";
// 读取返回流中的内容
while ((line = reader.readLine()) != null) {
    sBuffer.append(line).append("\r\n");
}
// 打印状态码
System.out.println("http response code=====>" + conn.
getResponseCode());
// 打印返回的内容
System.out.println("http response=====>" + sBuffer);
conn.disconnect();
} catch (MalformedURLException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

**【运行结果】**以上是通过 Java 的 net 包进行简单的 HTTP 请求的方法, 以上打印内容如图 3-3 所示, 输出 http 状态码是 200。返回的内容看起来很乱, 但是整理一下大家会发现, 这其实就是百度首页的 HTML 页面代码, 这些 HTML 代码经过浏览器的解析之后, 就成了我们平时看到的静态网页。



图 3-3 HTTP 请求结果截图



【代码解析】Java 中发出 HTTP 请求可以通过 `HttpURLConnection` 来实现，通过创建包含地址信息的 `URL` 对象获取 `HttpURLConnection` 对象，然后对 `HttpURLConnection` 对象进行连接信息的设置，再获取到输入流，就能接收到服务器传过来的响应内容。

在 HTTP 的响应中返回了一种状态码，在 HTTP 中定义了多种状态码，各自代表不同的含义，感兴趣的朋友可以自行搜索各种返回状态码所代表的含义。

标准的 HTTP 支持六种请求方法：`GET`、`POST`、`HEAD`、`PUT`、`DELETE`、`OPTIONS`，在使用标准 RESTful 规范的 Web 应用程序中，这六种方法都会用到，我们最常用的是 `GET` 和 `POST` 的请求方法。下面便通过代码来解释 `GET` 和 `POST` 请求的区别。

### 【范例】

客户端代码：

```
package com.hjc.http;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.URL;

public class HttpMethodDemo {
    public static void main(String[] args) {
        String path = "http://localhost:8080/demo/HttpServer";
        String result = "";
        try { // 捕获跑出的 IOException 异常
            result = httpPost(path);
            System.out.println(result);
            result = httpGet(path);
            System.out.println(result);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
```



```

* GET 方法 demo
* @param path
* @return
* @throws IOException
*/
public static String httpGet(String path) throws IOException {
    URL url = new URL(path + "?param1=hjc1&param2=hjc2");
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("GET");
    conn.connect();
    BufferedReader reader = new BufferedReader(new InputStreamReader(
        conn.getInputStream()));
    StringBuffer sBuffer = new StringBuffer();
    String line = "";
    while ((line = reader.readLine()) != null) {
        sBuffer.append(line).append("\r\n");
    }
    reader.close();
    return sBuffer.toString();
}

/**
* POST 方法 demo
* @param path
* @return
* @throws IOException
*/
public static String httpPost(String path) throws IOException {
    URL url = new URL(path);
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("POST");
    // HttpURLConnection 中的 doInput 默认为 true, 而 doOutput 默认为 false,
    所以如果需要写内容到流, 需要设置为 true
    conn.setDoOutput(true);
    conn.connect();
    // POST 方法不能通过 URL 传递参数, 需通过将参数写入 body 体来传递
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
        conn.getOutputStream()));
    writer.write("param1=hjc1&param2=hjc2");
    writer.close();
}

```

```

        BufferedReader reader = new BufferedReader(new InputStreamReader(
            conn.getInputStream()));
        StringBuffer sBuffer = new StringBuffer();
        String line = "";
        while ((line = reader.readLine()) != null) {
            sBuffer.append(line).append("\r\n");
        }
        reader.close();
        return sBuffer.toString();
    }
}

```

服务端代码:

在服务端我们简单地搭建一个 Tomcat 7 的 HTTP 服务器, 勾选项目的 Project Facets (从 properties 中进入) 中的 Dynamic Web Module, 项目下就会生成一个 web.xml, 在项目中添加一个 HttpServer 类, 并继承自 HttpServlet (需要从 Tomcat 7 的 lib 目录下复制一个 servlet-api.jar 添加到项目 buildPath 下), 或直接使用右键新建 Servlet, Eclipse 会自动生成一个 Servlet 类。

Servlet 类的代码如下:

```

package com.hjc.server;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/HttpServer")
public class HttpServer extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public HttpServer() {
    }

    protected void doGet(HttpServletRequest request,

```

```

        HttpServletResponse response) throws ServletException, IOException {
    System.out.println("get method");
    String param1 = request.getParameter("param1");
    String param2 = request.getParameter("param2");
    System.out.println("param1=" + param1);
    System.out.println("param2=" + param2);
    response.getWriter().write("get method from server");
}

protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
    System.out.println("post method");
    String param1 = request.getParameter("param1");
    String param2 = request.getParameter("param2");
    System.out.println("param1=" + param1);
    System.out.println("param2=" + param2);
    response.getWriter().write("post method from server");
}
}

```

然后在 web.xml 中添加此 Servlet 的部署(在 Tomcat 7 中使用 @WebServlet 注解配置即可,无须在 web.xml 中再配置):

```

<servlet>
    <servlet-name>HttpServer</servlet-name>
    <servlet-class>com.hjc.server.HttpServer</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name> HttpServer </servlet-name>
    <url-pattern>/ HttpServer </url-pattern>
</servlet-mapping>

```

**【运行结果】**以上就是搭建一个简单的 HTTP 服务器的过程,接收参数后打印,并在打印之后返回数据,客户端运行结果如下:

```

post method from server
get method from server

```

服务器端运行结果如下:

```

post method

```

```
param1=hjc1  
param2=hjc2  
get method  
param1=hjc1  
param2=hjc2
```

【代码解析】客户端发出 HTTP 的 GET 请求默认 `HttpURLConnection` 的设置即可，而 POST 请求，则需要设置调用 `HttpURLConnection` 的 `setRequestMethod` 方法为 POST。另外，因为 POST 消息必须通过 body 传递参数，因此需要调用 `setDoOutput` 为 true，并通过 `write` 方法写入参数到 body 体。服务端通过 `servlet` 即可实现，重写 `servlet` 的 `doGet` 和 `doPost` 方法来分别接受 GET 和 POST 请求。

### 3.1.4 Socket 编程

Socket（套接字）用于描述 IP 地址和端口，是一个通信链的句柄，可以用来实现不同虚拟机或不同计算机之间的通信。网络中的主机一般会运行多个服务器，每个服务器上的每一种服务都会打开一个 Socket 并绑定到一个端口，不同端口对应着不同的服务。IP 对应着网络上的计算机，而端口则对应着计算机上某个具体的进程或服务。就好比邮寄信件时，地址代表一个具体的居民房，人名对应着具体的居住其中的一个人，邮递员根据地址和人名，就能把信件准确地寄送出去。

根据连接启动的方式及本地套接字要连接的目标，套接字之间的连接可以分为服务器监听、客户端请求、连接确认三个步骤。

#### 1. 服务器监听

服务器端套接字只需要在程序启动之后处于等待连接的状态，并实时监控网络状态，等待其他客户端套接字的连接。

#### 2. 客户端请求

客户端套接字需要先创建一个套接字，并在套接字中描述服务器套接字的信息（IP 地址和端口号），然后再向服务器套接字提出连接请求。

#### 3. 连接确认

当服务器端套接字监听到或接收到客户端套接字的连接请求时，就会响应客户端套



接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，接收其他客户端套接字的连接请求。Socket 通信模型如图 3-4 所示（左边为服务端，右边为客户端）。

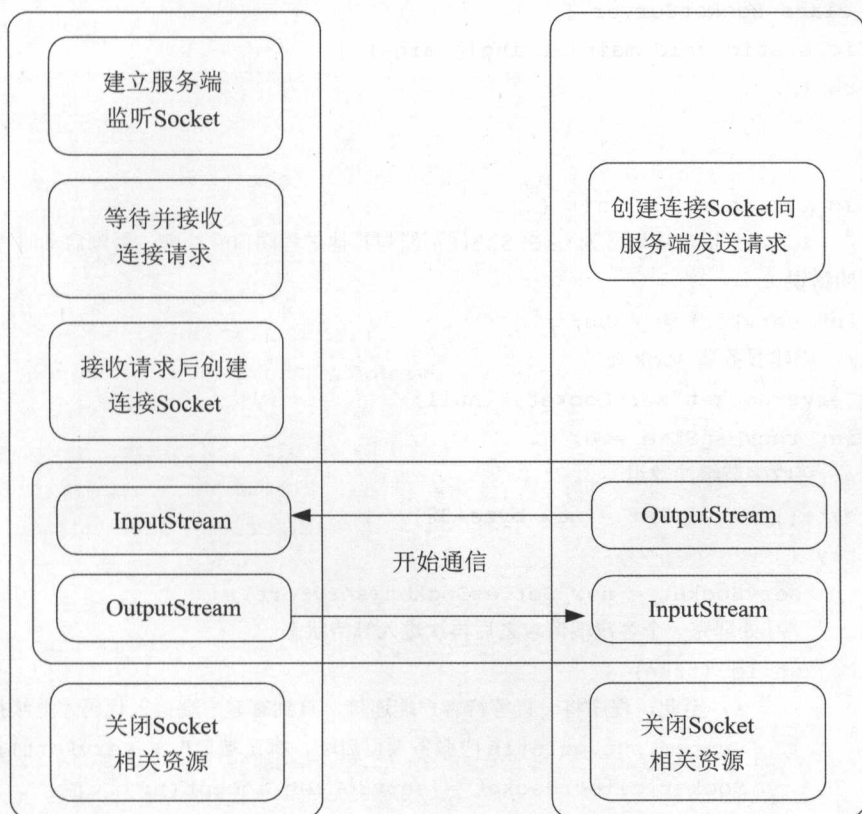


图 3-4 Socket 连接过程

IP 地址和端口号组成了 Socket，Socket 是网络上运行的程序之间双向通信链路的终结点，是 TCP 和 UDP 的基础。使用 Java 的 net 包中的 Socket API 可以实现基于 UDP 或 TCP 的 Socket 服务端与客户端。

### 【范例】

TCP 服务端：

```
package com.hjc.server;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
```

```
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;

public class SocketServer {
    public static void main(String[] args) {
        run();
    }

    public static void run() {
        // Socket 指定端口号为 0-65535, 不能与其他进程端口号冲突, 否则启动时程序会报
        // 端口占用的错误
        int servPort = 4700;
        // 创建服务端 Socket
        ServerSocket servSocket = null;
        int recvMsgSize = 0;
        // 接收字节缓冲数组
        byte[] receivBuf = new byte[32];
        try {
            servSocket = new ServerSocket(servPort);
            // 处理完一个客户端请求之后再次进入等待状态
            while (true) {
                // 至此, 程序将一直等待客户端连接, 直到有客户端接入代码才继续执行
                System.out.println("服务端已启动, 绑定端口" + servPort);
                Socket clientSocket = servSocket.accept();
                SocketAddress clientAddress = clientSocket
                    .getRemoteSocketAddress();
                System.out.println("收到客户端连接, ip: " + clientAddress);
                InputStream in = clientSocket.getInputStream();
                OutputStream out = clientSocket.getOutputStream();
                // 接收客户端发来的数据, 并原样返回给客户端
                while ((recvMsgSize = in.read(receivBuf)) != -1) {
                    String receivedData = new String(receivBuf.toString());
                    System.out.println(receivedData);
                    out.write(receivBuf, 0, recvMsgSize);
                }
                // 释放 Socket 资源
                clientSocket.close();
            }
        } catch (IOException e) {
```



```

        e.printStackTrace();
    }
}
}

```

### TCP 客户端:

```

package com.hjc.socket;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.SocketException;
import java.net.UnknownHostException;

public class SocketClientDemo {
    public static void main(String[] args) {
        String host = "localhost";
        int port = 4700;
        String sendMsg = "send data from client";
        connect(host, port, sendMsg.getBytes());
    }

    public static void connect(String server, int servPort, byte[] data) {
        // 创建 Socket 对象, 连接服务端 Socket
        Socket socket = null;
        try {
            socket = new Socket(server, servPort);
            System.out.println("连接服务器并发送数据...");
            InputStream in = socket.getInputStream();
            OutputStream out = socket.getOutputStream();
            out.write(data);
            // 接收数据
            // 目前收到的总字节长度
            int totalBytesRcvd = 0;
            // 最后一次读取的字节长度
            int bytesRcvd;
            // 将服务器返回消息读到 data 字节数组中
            while (totalBytesRcvd < data.length) {
                bytesRcvd = in.read(data, totalBytesRcvd, data.length

```

```

        - totalBytesRcvd);
    if (bytesRcvd == -1) {
        throw new SocketException("连接中断...");
    }
    totalBytesRcvd += bytesRcvd;
}
System.out.println("接收的数据:" + new String(data));
} catch (UnknownHostException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally { // 关闭 Socket 资源
    try {
        if (socket != null) {
            socket.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}
}

```

### 【运行结果】

TCP 服务端输出:

```

服务端已启动, 绑定端口 4700
收到客户端连接, ip: /127.0.0.1:52381
[B@73ea7821
服务端已启动, 绑定端口 4700

```

TCP 客户端输出:

```

连接服务器并发送数据...
接收的数据:send data from client

```

**【代码解析】**以上代码通过 `java.net` 包实现了 TCP 的客户端和服务端, 服务端使用 `ServerSocket` 绑定端口并启动, 然后通过阻塞方法 `accept` 接收并读取客户端发来的流信息, 而客户端则通过带套接字信息的 `Socket` 类连接服务端, 双方建立连接之后即可通过流进行双向传输。

## 【范例】

UDP 接收端:

```

package com.hjc.server;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

public class UDPSocketServer {
    public static void main(String[] args) {
        recive();
    }

    public static void recive() {
        System.out.println("接收端启动");
        // 接收端
        try {
            // 创建接收方的套接字对象,与 send 方法中 DatagramPacket 的 IP 地址与端口
            // 号一致
            DatagramSocket socket = new DatagramSocket(9001,
                InetAddress.getByName("localhost"));
            // 接收数据的 buf 数组并指定大小
            byte[] buf = new byte[1024];
            // 创建接收数据包,存储在 buf 中
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            // 接收操作,代码会停顿在这里,直到接收到数据包
            socket.receive(packet);
            byte data[] = packet.getData();// 接收的数据
            InetAddress address = packet.getAddress();// 接收的地址
            System.out.println("接收的文本==>" + new String(data));
            System.out.println("接收的 ip 地址==>" + address.toString());
            System.out.println("接收的端口==>" + packet.getPort()); // 9004
            // 告诉发送者接收完毕
            String temp = "接收端接收完毕了";
            byte buffer[] = temp.getBytes();
            // 创建数据报,指定发送给发送者的 socketaddress 地址
            DatagramPacket packet2 = new DatagramPacket(buffer, buffer.length,

```



```

        packet.getSocketAddress());
    // 发送
    socket.send(packet2);
    // 关闭
    socket.close();
} catch (SocketException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

### UDP 发送端:

```

package com.hjc.socket;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

public class UDPSocketClientDemo {

    public static void main(String[] args) {
        send();
    }

    // 发送者发给接收端数据, 接收端返回数据给发送者
    public static void send() {
        System.out.println("发送端发送数据");
        // 发送端
        try {
            // 创建发送方的套接字对象, 采用 9004 默认端口号
            DatagramSocket socket = new DatagramSocket(9004);
            // 发送的内容
            String text = "hello from sender!";
            byte[] buf = text.getBytes();
            // 构造数据包, 将长度为 length 的包发送到指定主机上的指定端口号
            DatagramPacket packet = new DatagramPacket(buf, buf.length,
                InetAddress.getByName("localhost"), 9001);

```

```

        // 套接字发送数据包
        socket.send(packet);
        // 接收者返回的数据
        displayRecvInfo(socket);
        // 关闭此数据报套接字
        socket.close();
    } catch (SocketException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

/**
 * 接收数据并打印出来
 *
 * @param socket
 * @throws IOException
 */
public static void displayRecvInfo(DatagramSocket socket)
    throws IOException {
    byte[] buffer = new byte[1024];
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    socket.receive(packet);

    byte data[] = packet.getData(); // 接收的数据
    InetAddress address = packet.getAddress(); // 接收的地址
    System.out.println("接收的文本==>" + new String(data));
    System.out.println("接收的 ip 地址==>" + address.toString());
    System.out.println("接收的端口==>" + packet.getPort()); // 9004
}
}

```

### 【运行结果】

#### UDP 接收端输出:

```

接收端启动
接收的文本==>hello from sender!
接收的 ip 地址==>/127.0.0.1
接收的端口==>9004

```

UDP 发送端输出:

```
发送端发送数据
接收的文本==>接收端接收完毕了
接收的 ip 地址==>/127.0.0.1
接收的端口==>9001
```

**【代码解析】**Java 也能很容易地实现 UDP 通信,服务端使用 `DatagramSocket` 类创建接收端套接字,客户端使用 `DatagramSocket` 类创建发送端套接字。因为 UDP 不是面向连接的,所以发送端创建套接字之后就可以开始发送数据包,数据包通过 `DatagramPacket` 进行封装,再调用 `DatagramSocket` 的 `send` 方法来发送数据包。

基于 TCP 和 UDP 的 `Socket` 代码实现如上所示,不过以上代码使用 BIO 进行通信,并不适合实际项目开发,后续章节会介绍 BIO 与其他 IO 模型的优缺点。

### 3.1.5 WebSocket 编程

WebSocket 是随着 HTML5 兴起的一种新协议。没有 WebSocket 之前,只能通过 HTTP 实现单通道通信,而 WebSocket 的出现实现了浏览器与服务器之间的全双工通信,使服务端主动推送更容易实现。在 WebSocket API 中,浏览器和服务器首先需要做一个“握手”动作,然后,浏览器和服务器之间就形成了一条快速通道,它们就可以直接传送数据。

这样的协议有着很小的 header,并且能实现服务端主动推送。在此之前,若要在 Web 开发中实现即时通信,一般会采用轮询 (polling) 方式,即浏览器在特定的时间间隔内不断发出 HTTP 请求来请求服务器的数据,然后服务端返回客户端请求时的最新数据给浏览器。但这样的模式存在很明显的缺陷:浏览器不断发送 HTTP 请求,并且每一个请求都带有很长的 header 头部信息,因此很占用带宽。而后来改善的长轮询方式,也和短轮询原理类似,只不过是服务端接收到 HTTP 请求之后不放开,使请求处于等待状态,直到有最新数据才返回这个请求,此方式并没有从根本上改善客户端与服务器的通信问题。

WebSocket 的出现恰好解决了浏览器与服务端需要全双工通信的需求,它的通信方式类似于 TCP 的连接方式,首先要进行“握手”协议通信,然后才能建立通信传输数据,这种方式能实现服务器对客户端的消息推送。



目前，支持 WebSocket 的浏览器如表 3-1 所示。

表 3-1 支持 WebSocket 的浏览器

Chrome	Supported in version 4+
Firefox	Supported in version 4+
Internet Explorer	Supported in version 10+
Opera	Supported in version 10+
Safari	Supported in version 5+

JavaEE 7 开始支持 WebSocket 规范，不少 Web 容器，如 Tomcat、Nginx、Jetty 等都支持 WebSocket。Tomcat 从 7.0.27 开始支持 WebSocket，因此以下 Demo 代码需要部署在 Tomcat 7 环境下运行。

### 【范例】

Java 服务端：

```
package com.hjc.server;

import java.io.IOException;

import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/websocket")
public class WebSocketServer {
    @OnMessage
    public void onMessage(String message, Session session) throws IOException,
        InterruptedException {
        // 接收客户端消息
        System.out.println("服务端接收：" + message);
        // 给客户端发送一条消息
        session.getBasicRemote().sendText("服务端发送第一条消息：" + message);
        // 服务端每 3s 给客户端主动推送一条消息
        int sentMessages = 0;
        while (sentMessages < 10) {
            Thread.sleep(3000);
            sentMessages++;
        }
    }
}
```

```

        session.getBasicRemote().sendText(
            "服务端主动推送的第" + sentMessages + "条消息");
    }
    // 给客户端发送最后一条消息
    session.getBasicRemote().sendText("这是服务端的最后一条消息");
    session.close();
}

@OnOpen
public void onOpen() {
    System.out.println("客户端建立连接");
}

@OnClose
public void onClose() {
    System.out.println("连接关闭");
}
}

```

自 J2EE7 开始才支持 WebSocket，因此以上代码需要部署在 Tomcat 7 上运行。

HTML 5 客户端：

```

<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>WebSocket 客户端</title>
</head>
<body>
<div>
    <input id="msg" type="text" value="" /> <input type="submit"
        value="发送" onclick="sendMsg()" />
</div>
<div id="messages"></div>
<script type="text/javascript">
    // websocket 协议访问以 ws 开头
    var websocket = new WebSocket('ws://localhost:8080/demo/websocket');
    // websocket 连接失败，回调函数
    websocket.onerror = function(event) {
        console.log("error:", event.data);
    }

```

```

};
// websocket 打开连接, 回调函数
websocket.onopen = function(event) {
    console.log("连接服务器成功");
};
// websocket 接收到消息, 回调函数
websocket.onmessage = function(event) {
    console.log("客户端接收消息:", event.data);
};
// websocket 断开连接, 回调函数
websocket.onclose = function(event){
    console.log("连接中断");
}
function sendMsg() {
    var msg = document.getElementById("msg").value;
    websocket.send("发送自客户端的消息:" + msg);
}
</script>
</body>
</html>

```

**【运行结果】**以上代码只有在支持 WebSocket 标准的浏览器环境下才能有效执行。我们用 chrome 打开写好的 html 页面, 然后通过 F12 键查看控制台, 在输入框输入要发送的内容, 再单击“发送”按钮, 服务端接收到浏览器发送的内容后会直接返回来, 并且每隔 3s 向浏览器推送一条消息, 推送满 10 条后关闭连接通道, 如图 3-5 所示。

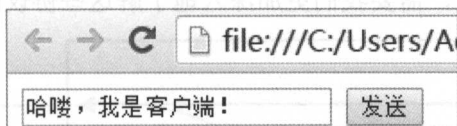


图 3-5 WebSocket 前端截图

服务端输出:

客户端建立连接

服务端接收: 发送自客户端的消息: 客户端消息

连接关闭

浏览器输出 (通过 F12 键查看 chrome 控制台):

websocketclient.html:22 连接服务器成功

websocketclient.html:26 客户端接收消息: 服务端发送第一条消息: 发送自客户端的消息:

哈喽，我是客户端！

```
websocketclient.html:26 客户端接收消息：服务端主动推送的第 1 条消息
websocketclient.html:26 客户端接收消息：服务端主动推送的第 2 条消息
websocketclient.html:26 客户端接收消息：服务端主动推送的第 3 条消息
websocketclient.html:26 客户端接收消息：服务端主动推送的第 4 条消息
websocketclient.html:26 客户端接收消息：服务端主动推送的第 5 条消息
websocketclient.html:26 客户端接收消息：服务端主动推送的第 6 条消息
websocketclient.html:26 客户端接收消息：服务端主动推送的第 7 条消息
websocketclient.html:26 客户端接收消息：服务端主动推送的第 8 条消息
websocketclient.html:26 客户端接收消息：服务端主动推送的第 9 条消息
websocketclient.html:26 客户端接收消息：服务端主动推送的第 10 条消息
websocketclient.html:26 客户端接收消息：这是服务端的最后一条消息
websocketclient.html:30 连接中断
```

**【代码解析】**服务端代码通过 Tomcat 7 实现，Tomcat 7 版本开始支持 WebSocket，通过注解 `@ServerEndpoint("/websocket")` 即可在 Tomcat 发布时发布为 WebSocket 端口。在服务端的 Java 类中，至少需要重写 `OnOpen`、`OnClose` 和 `OnMessage` 三个方法来分别实现 WebSocket 服务端的建立连接、断开连接和消息接收的方法，在消息接收的方法中即可实现自己的业务逻辑。客户端代码通过 HTML 5 实现，HTML 5 全面支持 WebSocket，在 HTML 5 中，同样实现 `OnOpen`、`OnClose` 和 `OnMessage` 方法即可。

上面分别介绍了 Http、Socket 及 WebSocket 在 Java 中代码的实现，不过这些都是在功能上简单的实现。在实际项目中，要根据实际情况进行开发，比如 http 响应超时设置、头部消息设置、Socket 或 WebSocket 的心跳机制、断线重连机制或多线程环境下的网络编程等。解决实际问题时，需要我们更加深入地了解这些协议的机制原理。

## 3.2 Java NIO 基础

Java I/O 类库正在不断发展和改进中，由最初的 BIO、NIO 到现在的 JDK1.7 的 NIO 2.0，基于 Java NIO 开发的网络服务器丝毫不逊色于 C++ 开发的网络服务器，Java 的网络编程也随着 NIO 越来越流行。

在讲解 Java 的几种网络编程模型之前，必须弄清楚同步和异步、阻塞和非阻塞的概念。同步指用户进程触发 IO 操作并等待或者轮询地去查看 IO 操作是否就绪。而异步是指用户进程触发 IO 操作以后便可以去做其他操作，当 IO 操作已经完成的时候会得到 IO



完成的通知。阻塞和非阻塞是进程在访问数据的时候，根据 IO 操作的就绪状态来采取的不同方式，实际上就是执行完数据操作之后是否立即返回值。同步和异步是指线程发起一个功能调用的时候是否立即返回结果，主要针对 IO 操作；而阻塞和非阻塞是指线程在得到调用结果之前是否被挂起，主要针对线程。

### 3.2.1 BIO 编程（Blocking-IO，阻塞式 IO）

BIO 属于同步阻塞型 IO，在服务器的实现模式为，每一个连接都要对应一个线程。当客户端有连接请求的时候，服务端需要启动一个新的线程与之对应进行处理。这个模式的缺陷很明显，当新创建的线程不做任何处理只是挂着时，就会给服务器造成不必要的线程开销。后来有人使用线程池来改善这种情况，但从本质上来说，问题仍然存在。这种通信模式要使用连接数目较少且固定的服务器架构，并且这种模式还要求服务器有较好的资源。它的并发仅仅局限于应用中，虽然有缺陷，但在 JDK1.4 之前，这种模式一直是 Java 网络编程的唯一方式，程序简单易理解。

在传统的网络编程中，服务器绑定 IP 并监听端口，客户端通过 IP 和端口与服务器建立连接，然后双方就可以通过 Socket 进行通信，ServerSocket 负责绑定 IP 并监听端口，Socket 负责发起连接操作，连接成功之后，双方就可以进行通信。在以上过程中，通常由一个接收器负责接入客户端的请求，然后创建一个新的线程，处理完成之后，再通过输出流返回给客户端，线程销毁。这种单一的“客户端—服务器”请求应答模型就是 BIO 的经典模型，我们可以通过图 3-6 来表示。

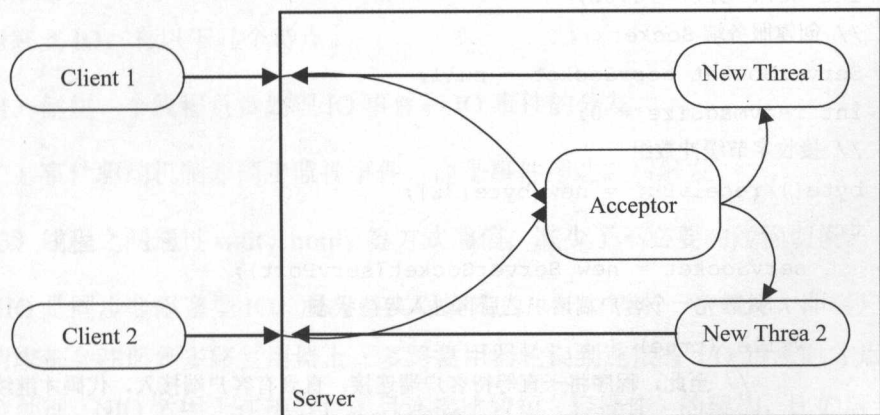


图 3-6 BIO 模型图

上图可以大概描述一个 BIO 网络的通信模型，这种 I/O 被称作同步阻塞式 I/O。这个模型的问题也显而易见，每一个客户端接入，服务器都要建立一个新的线程来进行处理，客户端连接数与服务器线程数以 1:1 的形式呈现。所以，当客户端连接数增多，产生高并发的时候，整个 BIO 网络程序将占用大量的 JVM 线程，造成服务器性能急剧下降，线程数达到一定数量后，JVM 会跑出堆栈溢出的异常，最终导致宕机。

### 【范例】

BIO 的代码 Demo 如下，与上文的 Socket 编程部分代码基本相同：

```
package com.hjc.io;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;

/**
 * BIO demo
 */
public class BioDemo {
    public static void main(String[] args) {
        // Socket 指定端口号为 0-65535，不能与其他进程端口号冲突，否则启动时程序会报
        // 端口占用的错误
        int servPort = 4700;
        // 创建服务端 Socket
        ServerSocket servSocket = null;
        int recvMsgSize = 0;
        // 接收字节缓冲数组
        byte[] receivBuf = new byte[32];
        try {
            servSocket = new ServerSocket(servPort);
            // 处理完一个客户端请求之后再进入等待状态
            while (true) {
                // 至此，程序将一直等待客户端连接，直到有客户端接入，代码才继续执行
                System.out.println("服务端已启动，绑定端口" + servPort);
                Socket clientSocket = servSocket.accept();
                SocketAddress clientAddress = clientSocket
```



```

        .getRemoteSocketAddress();
        System.out.println("收到客户端连接, ip: " + clientAddress);
        InputStream in = clientSocket.getInputStream();
        OutputStream out = clientSocket.getOutputStream();
        while ((recvMsgSize = in.read(receivBuf)) != -1) {
            String receivedData = new String(receivBuf.toString());
            System.out.println(receivedData);
            out.write(receivBuf, 0, recvMsgSize);
        }
        // 释放 Socket 资源
        clientSocket.close();
    }
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

【运行结果】同 TCP 通信运行结果一样。

【代码解析】上文中的 TCP 通信方式就是传统的 BIO 通信方式，ServerSocket 通过 accept 方法实现通信阻塞，直到有客户端的请求到达。

### 3.2.2 NIO 编程（Non-Blocking IO，非阻塞式 IO）

由于 BIO 有诸多缺陷，尤其是其在高并发环境下不可用，NIO 就应运而生了。NIO 是非阻塞式 IO，有以下几个特点。

- (1) 创建一个线程负责处理 IO 事件和 IO 事件的分发。
- (2) 事件驱动机制非同步监视事件，而是事件到达之后触发。
- (3) 线程之间通过 wait、notify 等方式通信，减少了不必要的线程切换。

NIO 是同步非阻塞型 IO，服务器实现模式为：一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有 I/O 请求时才启动一个线程进行处理。NIO 适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，自 JDK1.4 开始支持。

Java NIO 的原理可以用图 3-7 来描述。

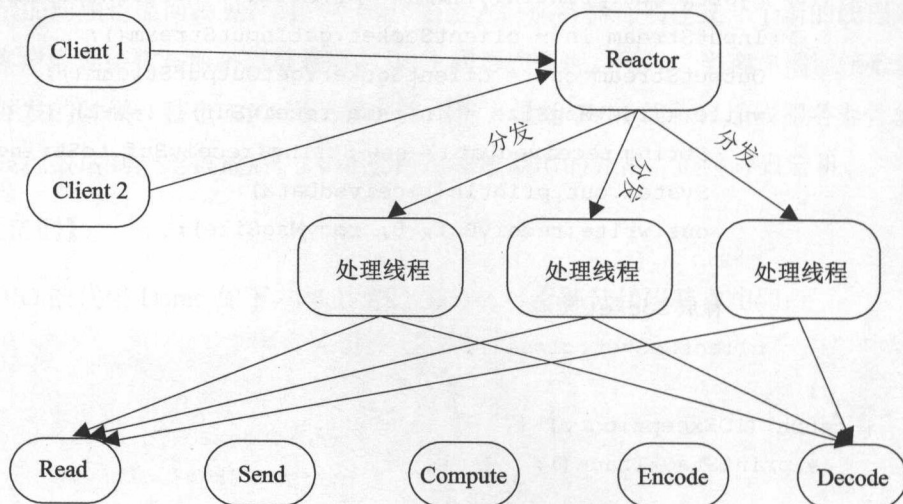


图 3-7 NIO 模型图

如上图所示，每一个线程处理的流程都相同：先读取数据，然后对数据进行编码，再进行计算处理，处理完之后再再响应消息编码，最后将响应发送出去。

Java NIO 的服务端只需启动一个专门的线程来处理所有的 IO 事件，这种通信模型的实现是因为其采用了双向通道（channel）进行数据传输，在通道上可以注册感兴趣的事件（服务端接收客户端连接事件，客户端连接服务端事件、读事件、写事件）。

NIO 的服务端和客户端均需要维护一个管理通道的对象（selector），该对象能检测一个或多个通道（channel）上的事件。如果服务端的 selector 上注册了读事件，某时刻客户端给服务端发送了一些数据，阻塞 I/O 会调用 read() 方法阻塞地读取数据，而 NIO 的服务端会在 selector 中添加一个读事件。服务端的处理线程会轮询地访问 selector，如果访问 selector 时发现感兴趣的事件到达，则处理这些事件；如果没有感兴趣的事件到达，则处理线程会一直阻塞到感兴趣的事件到达为止。

### 【范例】

服务端：

```
package com.hjc.socket;
```

```

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;

public class NioServer {
    // 通道管理器
    private Selector selector;

    /*
     * 获得一个 ServerSocket 通道, 并对该通道做一些初始化工作
     */
    * @param port 绑定的端口号
    *
    * @throws IOException
    */
    public void initServer(int port) throws IOException {
        // 获得一个 ServerSocket 通道
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        // 设置通道为非阻塞
        serverChannel.configureBlocking(false);
        // 将该通道对应的 ServerSocket 绑定到 port 端口
        serverChannel.socket().bind(new InetSocketAddress(port));
        // 获得一个通道管理器
        this.selector = Selector.open();
        /*
         * 将通道管理器和该通道绑定, 并为该通道注册 SelectionKey.OP_ACCEPT 事件
         * 注册该事件后, 当该事件到达时, selector.select() 会返回
         * 如果该事件没有到达, selector.select() 会一直阻塞
         */
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    }

    /*
     * 采用轮询的方式监听 selector 上是否有需要处理的事件, 如果有则进行处理
     */
}

```

```

    * @throws IOException
    */
    public void listen() throws IOException {
        System.out.println("服务端启动成功!");
        // 轮询访问 selector
        while (true) {
            // 当注册的事件到达时, 方法返回; 否则, 该方法会一直阻塞
            selector.select();
            // 获得 selector 中选中的项的迭代器, 选中的项为注册的事件
            Iterator ite = this.selector.selectedKeys().iterator();
            while (ite.hasNext()) {
                SelectionKey key = (SelectionKey) ite.next();
                // 删除已选的 key, 以防重复处理
                ite.remove();
                // 客户请求连接事件
                if (key.isAcceptable()) {
                    ServerSocketChannel server = (ServerSocketChannel) key
                        .channel();
                    // 获得和客户端连接的通道
                    SocketChannel channel = server.accept();
                    // 设置成非阻塞
                    channel.configureBlocking(false);
                    // 在这里可以给客户端发送信息
                    channel.write(ByteBuffer.wrap(new String("1234567890")
                        .getBytes()));
                    // 和客户端连接成功后, 为了接收到客户端的信息, 需要给通道设置读权限
                    channel.register(this.selector, SelectionKey.OP_READ);
                } else if (key.isReadable()) {
                    read(key);
                }
            }
        }
    }

    /*
    * 处理读取客户端发来的信息的事件
    *
    * @param key
    *
    * @throws IOException
    */

```



```

*/
public void read(SelectionKey key) throws IOException {
    // 服务器可读取消息：得到事件发生的 Socket 通道
    SocketChannel channel = (SocketChannel) key.channel();
    // 创建读取的缓冲区
    ByteBuffer buffer = ByteBuffer.allocate(10);
    channel.read(buffer);
    byte[] data = buffer.array();
    String msg = new String(data).trim();
    System.out.println("服务端收到信息: " + msg);
    ByteBuffer outBuffer = ByteBuffer.wrap(msg.getBytes());
    channel.write(outBuffer); // 将消息回送给客户端
}

public static void main(String[] args) throws IOException {
    AioServer server = new AioServer();
    server.initServer(8100);
    server.listen();
}
}

```

### 客户端:

```

package com.hjc.io;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Iterator;

public class NioClientDemo {
    // 通道管理器
    private Selector selector;

    /*
     * 获得一个 Socket 通道，并对该通道做一些初始化处理
     *
     * @param ip, 连接的服务器的 IP
     */
}

```

```

*
* @param port, 连接的服务器的端口号 @throws IOException
*/
public void initClient(String ip, int port) throws IOException {
    // 获得一个 Socket 通道
    SocketChannel channel = SocketChannel.open();
    // 设置通道为非阻塞
    channel.configureBlocking(false);
    // 获得一个通道管理器
    this.selector = Selector.open();
    /*
    * 客户端连接服务器, 其实方法执行并没有实现连接, 需要在 listen() 方法中调用
    channel.finishConnect() 才能完成连接
    */
    channel.connect(new InetSocketAddress(ip, port));
    // 将通道管理器和该通道绑定, 并为该通道注册 SelectionKey.OP_CONNECT 事件
    channel.register(this.selector, SelectionKey.OP_CONNECT);
}

/*
* 采用轮询的方式监听 selector 上是否有需要处理的事件, 如果有则进行处理
*
* @throws IOException
*/
public void listen() throws IOException {
    // 轮询访问 selector
    while (true) {
        /*
        * 选择一组可以进行 I/O 操作的事件, 放在 selector 中, 客户端的该方法不会阻塞,
        * 这里和服务端的方法不一样, 查看 api 注释可以知道, 当至少一个通道被选中时,
        * selector 的 wakeup 方法被调用, 方法返回, 而对于客户端来说, 通道一直是
        被选中的
        */
        selector.select();
        // 获得 selector 中选中的项的迭代器
        Iterator ite = this.selector.selectedKeys().iterator();
        while (ite.hasNext()) {
            SelectionKey key = (SelectionKey) ite.next();
            // 删除已选的 key, 以防重复处理
            ite.remove();
        }
    }
}

```



```

// 连接事件发生
if (key.isConnectable()) {
    SocketChannel channel = (SocketChannel) key.channel();
    // 如果正在连接, 则完成连接
    if (channel.isConnectionPending()) {
        channel.finishConnect();
    }
    // 设置成非阻塞
    channel.configureBlocking(false);
    // 给服务端发送信息
    channel.write(ByteBuffer.wrap(new String(
        "hello from client").getBytes()));
// 和服务端连接成功之后, 为了可以接收到服务端的信息, 需要给通道设置读权限
    channel.register(this.selector, SelectionKey.OP_READ);
    // 获得了可读事件
} else if (key.isReadable()) {
    read(key);
}
}

}

public void read(SelectionKey key) throws IOException {
    SocketChannel channel = (SocketChannel) key.channel();
    ByteBuffer buffer = ByteBuffer.allocate(10);
    channel.read(buffer);
    byte[] data = buffer.array();
    String msg = new String(data).trim();
    System.out.println("客户端收到信息: " + msg);
    ByteBuffer outBuffer = ByteBuffer.wrap(msg.getBytes());
    channel.write(outBuffer);
}

public static void main(String[] args) throws IOException {
    AioClientDemo client = new AioClientDemo();
    client.initClient("localhost", 8100);
    client.listen();
}
}

```

**【运行结果】**

服务端：

```
服务端启动成功！
服务端收到信息：hello from
服务端收到信息：client123
服务端收到信息：4567890
```

客户端：

```
客户端收到信息：1234567890
客户端收到信息：hello from
客户端收到信息：client123
```

**【代码解析】**Java 从 JDK1.4 开始支持 Java NIO，通过 `ServerSocketChannel` 类打开通道，对通道进行设置之后绑定到端口，将通道管理器和该通道绑定，并为该通道注册 `SelectionKey.OP_ACCEPT` 事件。注册该事件后，当该事件到达时，`selector.select()` 会返回，然后可以进行相应的处理，如果该事件没有到达，`selector.select()` 会一直阻塞。客户端的实现则通过 `SocketChannel` 调用 `connect` 方法与服务端进行连接，通过 `selector.select()` 方法获取并处理需要的事件。

### 3.2.3 AIO 编程（Async IO/NIO.2，异步 IO）

AIO 是异步非阻塞型 IO，在服务端的实现方式通常是一个有效的请求对应一个线程，而来自客户端的 I/O 请求都是由 OS 先完成再通知服务器应用去启动线程进行处理，适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂，JDK7 开始支持。以下是通过 NIO.2 实现的 Java 异步 I/O。

**【范例】**

服务端：

```
package com.hjc.server;

import java.io.IOException;
import java.net.InetSocketAddress;
```

```
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;

public class AioServer {

    static int PORT = 8200;
    static int BUFFER_SIZE = 1024;
    static String CHARSET = "utf-8"; // 默认编码
    static CharsetDecoder decoder = Charset.forName(CHARSET).newDecoder();
    // 解码

    int port;
    // ByteBuffer buffer;
    AsynchronousServerSocketChannel serverChannel;

    public AioServer(int port) throws IOException {
        this.port = port;
        // this.buffer = ByteBuffer.allocate(BUFFER_SIZE);
        AioServer.decoder = Charset.forName(CHARSET).newDecoder();
    }

    private void listen() throws Exception {
        // 打开一个服务通道, 绑定服务端口
        this.serverChannel = AsynchronousServerSocketChannel.open().bind(
            new InetSocketAddress(port), 100);
        this.serverChannel.accept(this, new AcceptorHandler());

        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    System.out.println("运行中...");
                    try {
                        Thread.sleep(2000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
        t.start();
    }
}
```



```

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

});
t.start();

}

/**
 * accept 到一个请求时的回调
 */
private class AcceptHandler implements
    CompletionHandler<AsynchronousSocketChannel, AioServer> {
    @Override
    public void completed(final AsynchronousSocketChannel client,
        AioServer attachment) {
        try {
            System.out.println("客户端地址: " + client.getRemoteAddress());
            // tcp 各项参数
            client.setOption(StandardSocketOptions.TCP_NODELAY, true);
            client.setOption(StandardSocketOptions.SO_SNDBUF, 1024);
            client.setOption(StandardSocketOptions.SO_RCVBUF, 1024);
            if (client.isOpen()) {
                System.out.println("客户端已打开: " + client.
getRemoteAddress());
                final ByteBuffer buffer = ByteBuffer.allocate (BUFFER_SIZE);
                buffer.clear();
                client.read(buffer, client, new ReadHandler(buffer));
            }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            attachment.serverChannel.accept(attachment, this); // 监听新
的请求, 递归调用
        }
    }
}

```

```

@Override
public void failed(Throwable exc, AioServer attachment) {
    try {
        exc.printStackTrace();
    } finally {
        attachment.serverChannel.accept(attachment, this); // 监听新
的请求, 递归调用
    }
}

/**
 * Read 到请求数据的回调
 */
private class ReadHandler implements
    CompletionHandler<Integer, AsynchronousSocketChannel> {

    private ByteBuffer buffer;

    public ReadHandler(ByteBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void completed(Integer result,
        AsynchronousSocketChannel attachment) {
        try {
            if (result < 0) { // 客户端关闭了连接
                AioServer.close(attachment);
            } else if (result == 0) {
                System.out.println("空数据"); // 处理空数据
            } else {
                // 读取请求, 处理客户端发送的数据
                buffer.flip();
                CharBuffer charBuffer = AioServer.decoder.decode(buffer);
                System.out.println(charBuffer.toString()); // 接收请求
                // 响应操作, 服务器响应结果
                buffer.clear();
                String res = "服务端返回的消息";
                buffer = ByteBuffer.wrap(res.getBytes());
            }
        }
    }
}

```

```

        attachment.write(buffer, attachment, new WriteHandler(
            buffer)); // Response: 响应
    }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

@Override
public void failed(Throwable exc, AsynchronousSocketChannel
attachment) {
    exc.printStackTrace();
    AioServer.close(attachment);
}

/**
 * Write 响应完请求的回调
 */
private class WriteHandler implements
    CompletionHandler<Integer, AsynchronousSocketChannel> {
    private ByteBuffer buffer;

    public WriteHandler(ByteBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void completed(Integer result,
        AsynchronousSocketChannel attachment) {
        buffer.clear();
        AioServer.close(attachment);
    }

    @Override
    public void failed(Throwable exc, AsynchronousSocketChannel attachment) {
        exc.printStackTrace();
        AioServer.close(attachment);
    }
}

```



```

public static void main(String[] args) {
    try {
        System.out.println("正在启动服务...");
        AioServer server = new AioServer(PORT);
        server.listen();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private static void close(AsynchronousSocketChannel client) {
    try {
        client.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

客户端代码直接使用 3.1.4 节的 TCP 客户端代码，将其中的端口号改为 AioServer 使用的 8200:

```

package com.hjc.socket;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.SocketException;
import java.net.UnknownHostException;

public class TCPSocketClientDemo {
    public static void main(String[] args) {
        String host = "localhost";
        int port = 8200;
        String sendMsg = "send data from client";
        connect(host, port, sendMsg.getBytes());
    }
}

```

```
public static void connect(String server, int servPort, byte[] data) {  
    // 创建 socket 对象, 连接服务端 socket  
    Socket socket = null;  
    try {  
        socket = new Socket(server, servPort);  
        System.out.println("连接服务器并发送数据...");  
        InputStream in = socket.getInputStream();  
        OutputStream out = socket.getOutputStream();  
        out.write(data);  
        // 接收数据  
        // 目前收到的总字节长度  
        int totalBytesRcvd = 0;  
        // 最后一次读取的字节长度  
        int bytesRcvd;  
        // 将服务器返回消息读到 data 字节数组中  
        while (totalBytesRcvd < data.length) {  
            bytesRcvd = in.read(data, totalBytesRcvd, data.length  
                                - totalBytesRcvd);  
            if (bytesRcvd == -1) {  
                throw new SocketException("连接中断...");  
            }  
            totalBytesRcvd += bytesRcvd;  
        }  
        System.out.println("接收的数据:" + new String(data));  
    } catch (UnknownHostException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally { // 关闭 socket 资源  
        try {  
            if (socket != null) {  
                socket.close();  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

**【运行结果】**

服务端:

```
正在启动服务...
运行中...
运行中...
运行中...
运行中...
客户端地址: /127.0.0.1:50904
客户端已打开: /127.0.0.1:50904
send data from client
运行中...
运行中...
```

客户端:

```
连接服务器并发送数据...
接收的数据:服务端返回的消息
```

**【代码解析】**Java 从 JDK7 开始支持 AIO 编程,服务端实现通过 `AsynchronousServerSocketChannel` 绑定端口,然后调用 `accept` 方法绑定一个消息处理类,这个类需要继承自 `CompletionHandler`,并重写其中需要处理的方法。客户端则直接使用 TCP 客户端代码。

由最后的打印结果也可以看出,AioServer 启动之后并没有阻塞在一直等待客户端连接的状态,而是可以做其他事情(如上面的代码是线程每休眠 2 秒打印一个“运行中...”)。由此也能看出 AIO 编程中,服务端无须等待 IO 全部处理完成再去执行其他的操作,只需要将 IO 操作扔给 OS,就可以去做别的操作,操作完成之后系统会回调通知结果,这样大大节省了系统的内存资源。

### 3.3 Mina 的介绍及其使用

Apache Mina 是基于 Java NIO 的网络应用框架,它的底层实现是基于 TCP/IP 和 UDP/IP 的。Mina 是一款协议栈的通信框架,使用 Mina 可以快速开发高性能、高扩展性的网络应用。Mina 提供了事件驱动、异步操作的模型,它的 IO 默认以 Java NIO 作为底

层支持，与 Netty 设计理念极其相似，都有着优雅的架构。这两款强大的网络框架并不是三言两语就能够说清楚的，我们只对其使用做简单的介绍，能够上手使用即可，可以去官方论坛、社区等进行更加深入的学习。Mina 从诞生到现在，已经有了很多版本，各方面都在不断的优化中。学习这款网络框架之前，必须要掌握 Java 网络编程相关知识，包括 Java IO、Java NIO、Java Socket、Java 线程与并发等知识，Mina 提供了很多功能的封装。

学习一款框架，需要从它的原理及工作模型、模块组件进行了解。下面我们就分别从总体架构、IoService、IoFilterChain、IoHandler、IoSession、工作原理、线程模型等方面介绍 Mina 的大致结构。

### 3.3.1 总体架构

Mina 的总体架构如图 3-8 所示。

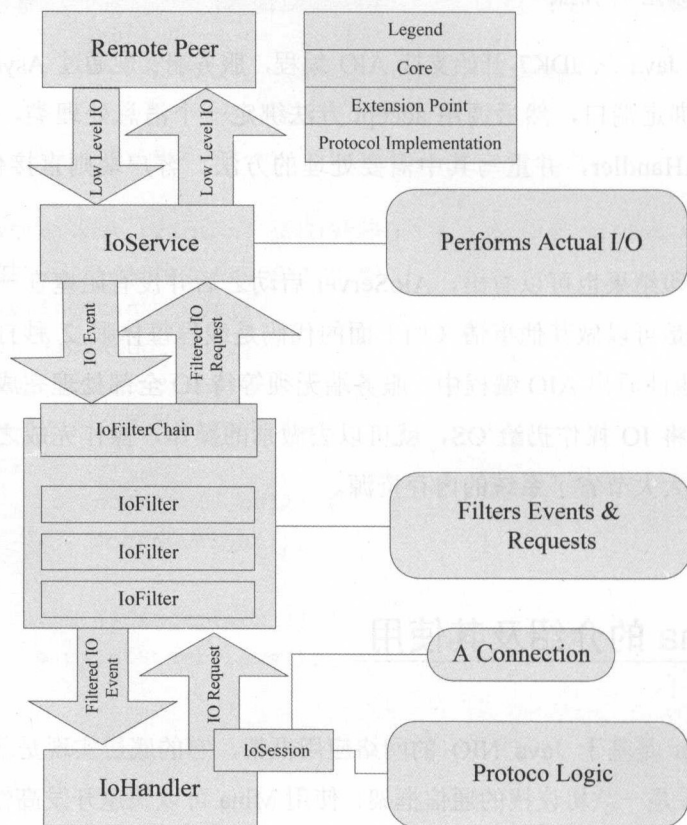


图 3-8 Mina 总体架构图



### 3.3.2 IoService

IoService 处于 Mina 的最底层，负责 Mina 底层的 IO 的相关工作。最典型的代表是 IoSocketAcceptor 和 IoSocketChannel，分别对应 TCP 下的服务端与客户端的 IoService。IoService 隐藏了所有底层的细节，对上层只提供统一的基于事件的异步 IO 接口。当有数据到达时，IoService 会先调用底层 IO 接口读取数据，并封装为 IoBuffer，然后再以事件的形式通知上层代码，将 Java NIO 的同步 IO 接口转换成异步 IO。

### 3.3.3 IoFilterChain

代码分离是 Mina 的设计之一，Mina 的业务代码和数据处理分别负责不同的内容，Mina 就把它们进行了分离。在 Mina 中，开发者开发的业务代码只需要专注于业务逻辑，而其他无关的逻辑，如数据包的解析、封装和过滤等，则交给 IoFilterChain 来处理。IoFilterChain 是 Mina 处理流程的扩展点，通过扩展 IoFilterChain，可以使 Mina 的结构划分更清晰，代码分工更明确。开发者如果想要增加处理流程而不影响后续的业务逻辑代码，只需要向 Chain 中添加 IoFilter 即可。

### 3.3.4 IoHandler

在 Mina 中要实现的业务逻辑，都是在 IoHandler 中完成的。开发过程中，开发者需要自己实现 IoHandler 接口。IoHandler 是 Mina 处理流程的终点，每个 IoService 都要指定一个 IoHandler。

### 3.3.5 IoSession

Mina 将底层的连接封装成 IoSession，每个 IoSession 都对应一个客户端与服务端的底层 IO 连接，因此，通常开发者需要管理好每一个客户端的 IoSession。通过 IoSession，不但可以获取连接的相关信息，更可以向客户端发送数据。通过 IoSession 发送数据是一个异步过程，发送操作首先通过 IoFilterChain 到达 IoService，然后 IoService 将发送操作封装成 WriteRequest，并放入 Session 的 writeRequestQueue 中，最后交由 IoProcessor 统一调度 flush 发送出去，整个发送操作不会引起调用线程的阻塞。



### 3.3.6 工作原理

Mina 使用 Java NIO 的线程调度原理如图 3-9 所示。

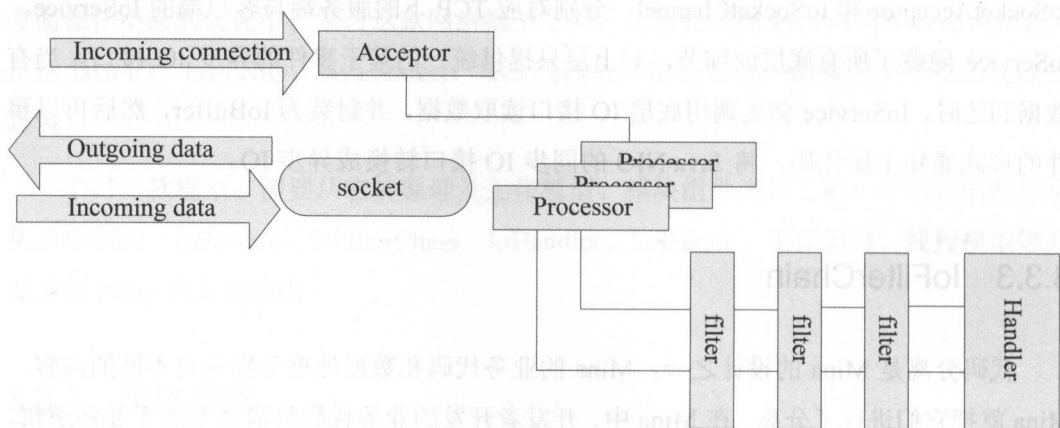


图 3-9 Mina 工作原理图

### 3.3.7 Acceptor 与 Connector 线程

在服务端启动并绑定端口之后，Mina 就创建 Acceptor 线程，专门负责监听，这个线程的工作就是调用 Java NIO 接口在该端口上的 select connect 事件，在获取到新建的连接之后，就会将其封装成 IoSession，并交由 IoProcessor 线程处理。在客户端也会有一个类似的对应的 Connector 线程，这两个线程是一对一的关系。一旦建立之后，外界就无法控制其线程的数量，直到连接断开。

### 3.3.8 Processor 线程

Processor 线程主要负责 IO 读写操作与执行 IoFilterChain 和 IoHandler 逻辑，它默认会有 CPU 数量+1 个线程，并且这个数量是可以通过配置参数进行控制的。每一个进来的 IoSession 都会被分配到这些线程中，默认策略是 session id 绝对值对 N 取模来分配。

Processor 线程维护着一个 selector，并对维护着的 IoSession 进行 select 和遍历，然后读取数据，以事件的形式通知 IoFilterChain，并对请求队列进行 flush 操作。

把 `IoSession` 均分到多个 `Processor` 线程中进行处理, 可以充分利用计算机多核的处理能力, 减轻 `select` 操作的压力。虽然默认的 `Processor` 的线程数量能够满足大部分情况下的需求, 但在实际项目中, 还需要根据实际线上环境进行测试和修改。

### 3.3.9 线程模型

从每一个 `Processor` 的线程内部看, `IO` 请求的处理是按顺序处理的, 也就是在 `Processor` 中是按照单线程进行处理。上文提过, 当 `Process` 线程 `select` 到一批就绪的 `IO` 请求之后, 就会在线程内部遍历并一一对这些 `IO` 请求进行处理。处理的流程包括 `IoFilter` 和 `IoHandler` 里的逻辑。

当就绪 `IO` 前面的 `IO` 请求处理完之后, 才会再取出下一个 `IO` 请求进行处理。所以, 如果 `IoFilter` 或 `IoHandler` 中有比较耗时的操作, 比如读取数据库等, `Processor` 线程将会被阻塞住, 后续的请求将不会被处理。显然, 有高并发需求的服务器是不能容忍这种情况的。所以, `Mina` 通过在处理流程中引入线程池来解决这个问题。

在 `Mina` 中, 通过 `IoFilter` 的形式为处理流程添加线程池。`Mina` 的线程模型主要有如图 3-10 所示的几种形式。

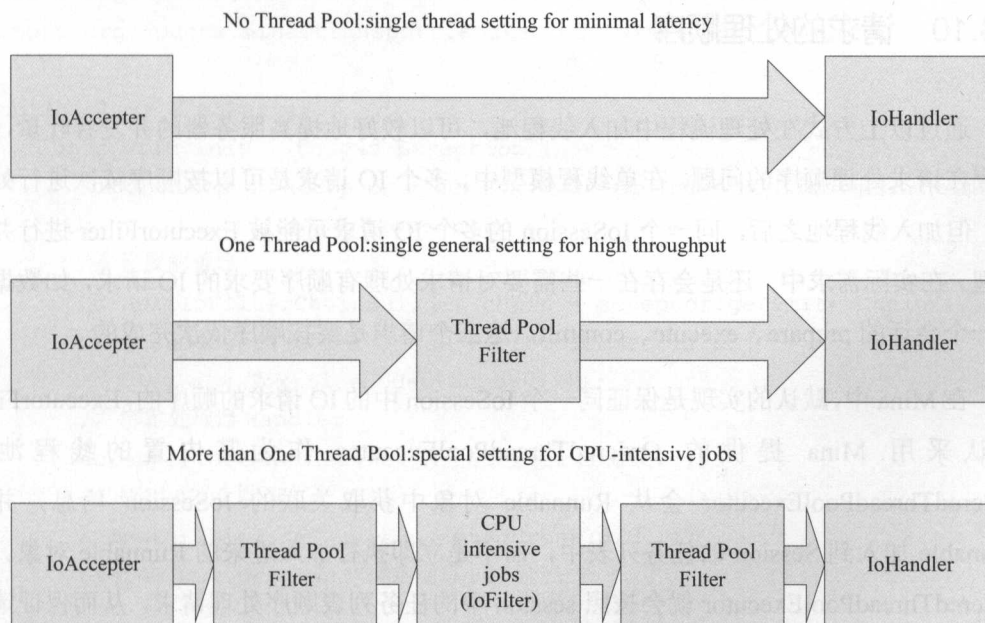


图 3-10 `Mina` 线程模型

Mina 中常见的线程模型如下。

### 1. 单线程模型

Mina 默认线程模型, Processor 处理了从底层 IO 到上层 IoHandler 逻辑所执行的所有工作, 这种模式适合逻辑不复杂且能快速返回的情况。

### 2. 单线程池模型

在 IoFilterChain 中加入 Thread Pool Filter, 当 Processor 线程读取完数据之后, 执行 IoFilterChain 逻辑, 当进行到 Thread Pool Filter 的时候, 此 Filter 将后续处理封装至 Runnable 中, 并交给 Filter 自身的线程池执行, Processor 则立即返回处理下一个 IO 请求。这种流程模式下, 如果在 IoFilter 或 IoHandler 中出现了阻塞操作, 只会让执行此操作的 Filter 阻塞, 而不会阻塞负责读取数据的 Processor 线程, 从而提高服务器的并发及处理能力。Mina 中提供了 Thread Pool Filter 的实现: ExecutorFilter。

### 3. 多线程池模型

多线程池模型是在上面两种模型的基础上, 将单个 Thread Pool Filter 变为多个 Thread Pool Filter。

## 3.3.10 请求的处理顺序

通过以上方式在处理流程中加入线程池, 可以较好地提高服务器的并发吞吐量, 但也存在请求处理顺序的问题。在单线程模型中, 多个 IO 请求是可以按顺序依次进行处理的, 但加入线程池之后, 同一个 IoSession 的多个 IO 请求可能被 ExecutorFilter 进行并行处理。在实际需求中, 还是会存在一些需要对请求处理有顺序要求的 IO 请求, 如数据库同一个会话的 prepare、execute、commit, 这三个请求是要按顺序依次完成的。

在 Mina 中, 默认的实现是保证同一个 IoSession 中的 IO 请求的顺序的, ExecutorFilter 默认采用 Mina 提供的 OrderedThreadPoolExecutor 作为其内置的线程池。OrderedThreadPoolExecutor 会从 Runnable 对象中获取关联的 IoSession 信息, 并将 Runnable 加入到 Session 的任务列表中, 而不是立即执行加入进来的 Runnable 对象。而 OrderedThreadPoolExecutor 就会按照 session 中的任务列表顺序处理请求, 从而保证请求的执行顺序。

当然,对于没有执行顺序要求的IO请求情景,可以为ExecutorFilter指定一个Executor来代替默认的OrderedThreadPoolExecutor,这样就能实现多个session被并行处理,以及提高服务器并发吞吐量。

### 3.3.11 Mina 编程

Mina 框架良好地封装了 Java NIO 的相关底层,使 Java 在 Mina 的基础上对网络的编程更加容易。下面讲解在 Java 中使用 Mina 的实例。

#### 【范例】

服务端:

```
package com.hjc.demo.mina;

import java.net.InetSocketAddress;

import org.apache.mina.core.filterchain.DefaultIoFilterChainBuilder;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.transport.socket.SocketAcceptor;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaServer {
    public void init() throws Exception {
        SocketAcceptor acceptor = new NioSocketAcceptor(Runtime.getRuntime()
            .availableProcessors() + 1);
        // 设置解析器
        DefaultIoFilterChainBuilder chain = acceptor.getFilterChain();
        chain.addLast("codec", new ProtocolCodecFilter(
            new TextLineCodecFactory()));
        // 绑定处理器 Handler
        acceptor.setHandler(new MinaServerHandler());
        // 绑定 8200 端口
        acceptor.bind(new InetSocketAddress(8200));
        System.out.println("绑定端口 8200");
    }

    public MinaServer() throws Exception {
```

```
        init();
    }

    public static void main(String[] args) throws Exception {
        new MinaServer();
        System.out.println("Mina 服务器开启");
    }
}
```

### Mina 服务端的消息处理类:

```
package com.hjc.demo.mina;

import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.session.IoSession;

public class MinaServerHandler extends IoHandlerAdapter {
    @Override
    public void messageReceived(IoSession session, Object message)
        throws Exception {
        session.write("服务端发送消息 " + message);
        session.write("msg from server");
        session.write("quit");
        // session.close(true);
    }

    @Override
    public void exceptionCaught(IoSession session, Throwable cause)
        throws Exception {
        if (session.isConnected()) {
            session.close(true);
        }
    }

    @Override
    public void messageSent(IoSession session, Object message) throws Exception {
        System.out.println("服务端收到的消息: " + message.toString());
        // session.close(true);
    }
}
```



```

@Override
public void sessionClosed(IOException session) throws Exception {
    super.sessionClosed(session);
    System.out.println("sessionClosed");
}

@Override
public void sessionCreated(IOException session) throws Exception {
    session.getConfig().setIdleTime(IdleStatus.BOTH_IDLE, 30000);
}

@Override
public void sessionIdle(IOException session, IdleStatus status)
    throws Exception {
    session.close(true);
}

@Override
public void sessionOpened(IOException session) throws Exception {
    super.sessionOpened(session);
}
}

```

客户端:

```

package com.hjc.demo.mina;

import java.net.InetSocketAddress;

import org.apache.mina.core.filterchain.DefaultIoFilterChainBuilder;
import org.apache.mina.core.future.ConnectFuture;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.transport.socket.nio.NioSocketConnector;

public class MinaClient {

    public static void run(int index) {
        // 创建 TCP/IP 连接
        NioSocketConnector connector = new NioSocketConnector();
        // 创建接收数据的过滤器
    }
}

```

```

DefaultIoFilterChainBuilder chain = connector.getFilterChain();
// 设定过滤器一行一行 (/r/n) 地读取数据
chain.addLast("myChin", new ProtocolCodecFilter(
    new TextLineCodecFactory()));
// 服务器的消息处理器: 一个 MinaClientHandler 对象
connector.setHandler(new MinaClientHandler(index));
// set connect timeout
connector.setConnectTimeout(30);
// 连接到服务器
ConnectFuture cf = connector.connect(new InetSocketAddress("localhost",
    8200));
cf.awaitUninterruptibly();
cf.getSession().getCloseFuture().awaitUninterruptibly();
connector.dispose();

}

public static void main(String[] args) throws Exception {
    for (int i = 0; i < 5; i++) { // 运行 10 个客户端
        run(i + 1);
    }
}
}

```

Mina 客户端的消息处理类:

```

package com.hjc.demo.mina;

import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.session.IoSession;

public class MinaClientHandler extends IoHandlerAdapter {
    int index = 0;

    public MinaClientHandler(int index) {
        this.index = index;
    }

    @Override
    public void exceptionCaught(IoSession arg0, Throwable arg1)

```

```

        throws Exception {

    }

    /**
     * 当客户端接收到消息时
     */
    @Override
    public void messageReceived(IOException session, Object message)
        throws Exception {
        // 我们已设定了服务器的消息规则是一行一行地读取, 这里就可以转为 String
        String s = (String) message;
        System.out.println("客户端" + index + "收到的消息: " + s);
        if (s.equals("quit")) {
            session.close(true);
        }
        // 测试将消息回送给客户端
        session.write("客户端" + index + "收到了" + s);
    }

    @Override
    public void messageSent(IOException arg0, Object arg1) throws Exception {
        System.out.println("客户端" + index + "发送的消息: " + arg1.toString());
    }

    /**
     * 当一个客户端被关闭时
     */
    @Override
    public void sessionClosed(IOException session) throws Exception {
        System.out.println("客户端 " + session.getRemoteAddress() + " 断开");
    }

    @Override
    public void sessionCreated(IOException arg0) throws Exception {

    }

    @Override

```



```

public void sessionIdle(ioSession arg0, IdleStatus arg1) throws Exception {

}

/**
 * 当一个客户端连接进入时
 */
@Override
public void sessionOpened(ioSession session) throws Exception {
    session.write("客户端 " + session.getRemoteAddress() + " 连接打开!");
}

}

```

## 【运行结果】

服务端：

绑定端口 8200

Mina 服务器开启

服务端收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

服务端收到的消息：msg from server

服务端收到的消息：quit

服务端收到的消息：服务端发送消息 客户端 1 收到了服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

服务端收到的消息：msg from server

服务端收到的消息：quit

sessionClosed

服务端收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

服务端收到的消息：msg from server

服务端收到的消息：quit

服务端收到的消息：服务端发送消息 客户端 2 收到了服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

服务端收到的消息：msg from server

sessionClosed

服务端收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

服务端收到的消息：msg from server

sessionClosed

服务端收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

服务端收到的消息：msg from server

```
sessionClosed
```

服务端收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

服务端收到的消息：msg from server

```
sessionClosed
```

### 客户端：

客户端 1 发送的消息：客户端 localhost/127.0.0.1:8200 连接打开！

客户端 1 收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

客户端 1 发送的消息：客户端 1 收到了服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！ 客户端 1 收到的消息：msg from server

客户端 1 收到的消息：quit

客户端 1 发送的消息：客户端 1 收到了 msg from server

客户端 localhost/127.0.0.1:8200 断开

客户端 2 发送的消息：客户端 localhost/127.0.0.1:8200 连接打开！

客户端 2 收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

客户端 2 发送的消息：客户端 2 收到了服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

客户端 2 收到的消息：msg from server

客户端 2 收到的消息：quit

客户端 2 发送的消息：客户端 2 收到了 msg from server

客户端 localhost/127.0.0.1:8200 断开

客户端 3 发送的消息：客户端 localhost/127.0.0.1:8200 连接打开！

客户端 3 收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

客户端 3 发送的消息：客户端 3 收到了服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

客户端 3 收到的消息：msg from server

客户端 3 收到的消息：quit

客户端 3 发送的消息：客户端 3 收到了 msg from server

客户端 localhost/127.0.0.1:8200 断开

客户端 4 发送的消息：客户端 localhost/127.0.0.1:8200 连接打开！

客户端 4 收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

客户端 4 发送的消息：客户端 4 收到了服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

客户端 4 收到的消息：msg from server

客户端 4 收到的消息：quit

客户端 4 发送的消息：客户端 4 收到了 msg from server

客户端 localhost/127.0.0.1:8200 断开

客户端 5 发送的消息：客户端 localhost/127.0.0.1:8200 连接打开！

客户端 5 收到的消息：服务端发送消息 客户端 localhost/127.0.0.1:8200 连接打开！

客户端 5 发送的消息：客户端 5 收到了服务端发送消息 客户端 localhost/127.0.0.1:8200



连接打开!

客户端 5 收到的消息: msg from server

客户端 5 收到的消息: quit

客户端 5 发送的消息: 客户端 5 收到了 msg from server

客户端 localhost/127.0.0.1:8200 断开

**【代码解析】**基于 Mina 框架进行开发,只需要理解 Mina 的设计思想和其使用的 API 即可。Mina 服务端的实现需要先实例化一个 `SocketAcceptor`, 然后设置相应的解析器, 绑定处理器 `Handler`, 再绑定端口即可。在 `Handler` 中, 重写需要的方法进行相应的逻辑处理。在 Mina 客户端开发中, 需要先创建 `NioSocketConnector` 对象, 然后设置过滤器和处理器 `Handler`, 最后调用 `connect` 方法建立连接, 也是在 `Handler` 处理相应的业务逻辑。Mina 可通过调用 `IoSession` 的 `write` 方法发送消息。

通过简单的 Demo 示例, 我们基本可以简单地使用 Mina 进行 Java 网络程序的客户端或服务端的开发。接下来将介绍与 Mina 相差无几的一款 NIO 框架——Netty, 它们的实现原理及底层框架是大致相同的, 希望读者了解之后, 能在实际应用中根据需求选择最适合项目需求的网络层框架。

## 3.4 Netty 的介绍及其使用

Netty 源自 JBoss 的 Java 开源框架, 是一款提供异步事件驱动的网络应用框架, 主要用于开发高效、可靠的网络程序或客户端程序。随着版本迭代, Netty 已经脱离了 JBoss, 成为了独立的应用框架。

早期的网络编程是异常烦琐和复杂的, 而 Netty 实现了网络应用开发的简单化与流线性化, 如 TCP 或 UDP 的 `Socket` 服务器开发。Netty 开发的应用程序, 通常不会让用户在后期对性能进行维护。Netty 吸收了多种协议的经验, 包括但不限于 FTP、SMTP、HTTP、其他二进制协议、其他文本协议等。Netty 经过相当精细的设计, 形成了一套既易于开发, 又能保证应用的性能。

### 3.4.1 总体架构

Netty 总体架构如图 3-11 所示。

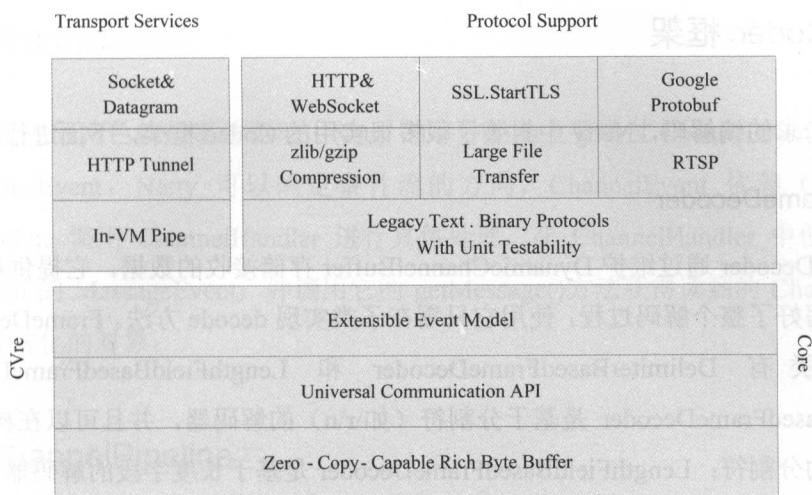


图 3-11 Netty 架构图

### 3.4.2 零拷贝

Netty 中的零拷贝主要体现在以下几个方面。

(1) Netty 的接收和发送采用 ByteBuffer，ByteBuffer 采用 Direct Buffers，即 ByteBuffer 直接使用堆外的内存进行 Socket 读写，而不需要进行字节缓冲区的二次拷贝。相比传统的使用堆内存（Heap Buffers）的 Socket 读写，JVM 再拷贝一份 Buffer 到内存中，然后再写入 Socket，会少一次缓冲区的内存拷贝。

(2) Netty 的组合 Buffer 对象，能聚合多个 ByteBuffer 对象，用户如果要操作多个 Buffer，可以先将这些 Buffer 组合，然后操作这个组合 Buffer，而传统的操作多个 Buffer 只能通过内存拷贝的方式将多个 Buffer 组合成一个大的 Buffer，这样更节省内存。

(3) Netty 采用 transferTo 进行文件传输，可以直接把文件缓冲区的数据发送到目标的 Channel，从而避免传统方式通过循环 write 导致的内存拷贝问题。

Netty 不使用 NIO 的 ByteBuffer，而是使用自建的 Buffer API，来表示一个连续的字节序列，因此，自建 ByteBuffer 会有更明显的优势。Netty 的新 Buffer 类型 ChannelBuffer 被设计为一个能从底层解决 ByteBuffer 内存的问题，可满足大多数网络应用中需要的缓冲类型，并且 ByteBuffer 允许自定义缓冲类型。ByteBuffer 还可以通过内置的缓冲类型实现透明的零拷贝，并且提供开箱即用的动态缓冲类型，其容量还可以根据需求进行扩充。综上所述，Netty 的自建 ByteBuffer 通常比 Java NIO 的 ByteBuffer 快。

### 3.4.3 Codec 框架

对于请求的编解码，Netty 中封装了很多很实用的 Codec 框架，下面进行简单介绍。

#### 1. FrameDecoder

FrameDecoder 通过维护 DynamicChannelBuffer 存储接收的数据，它提供抽象模板，在模板中写好了整个解码过程，使用它只需在子类实现 decode 方法。FrameDecoder 直接的实现类有 DelimiterBasedFrameDecoder 和 LengthFieldBasedFrameDecoder。DelimiterBasedFrameDecoder 是基于分割符（如\r\n）的解码器，并且可以在构造函数中指定自己的分割符；LengthFieldBasedFrameDecoder 是基于长度字段的解码器。

#### 2. ReplayingDecoder

ReplayingDecoder 是 FrameDecoder 的非阻塞解码，如果 FrameDecoder 读到的数据是不完整的，使用 ReplayingDecoder 就可以假设已经读取到了全部数据。

#### 3. ObjectEncoder 和 ObjectDecoder

这两个类能对 Java 对象进行编解码序列化。

#### 4. HttpRequestEncoder 和 HttpRequestDecoder

Netty 中还能实现 HTTP 服务器，通过 HttpRequestEncoder 和 HttpRequestDecoder 能实现 HTTP 请求和响应的编解码。

### 3.4.4 Channel

Channel 在 Netty 中主要实现以下功能。

(1) Channel 记录了当前的状态信息，即 Channel 为打开或关闭。

(2) 通过 ChannelConfig 可以得到 Channel 的配置信息。

(3) Channel 支持 read、write、bind、connect 等操作。

(4) 通过 Channel 可以得到处理该 Channel 的 ChannelPipeline。

Netty 通过 NioServerSocketChannel 封装 ServerSocketChannel、NioSocketChannel 封装 SocketChannel，实现了 Java NIO 中 Channel 的功能。

### 3.4.5 ChannelEvent

Netty 最大的特色就是事件驱动，Netty 的事件驱动主要通过 ChannelEvent 来实现。通过 ChannelEvent，Netty 可以确定事件流的方向，ChannelEvent 依赖 Channel 的 ChannelPipeline 调用 ChannelHandler 进行具体处理，在 ChannelHandler 中使用继承自 ChannelEvent 的 MessageEvent，并调用它的 getMessage()方法获得读到的 ChannelBuffer 或者已经被转化的对象。

### 3.4.6 ChannelPipeline

Netty 控制事件流是通过 ChannelPipeline 处理的，通过调用在 ChannelPipeline 中注册的一系列 ChannelHandler 来处理事件，这其实是很典型的拦截器模式。事件流分为 Upstream 和 Downstream 事件（即上行流和下行流）。在 ChannelPipeline 中，被注册的 ChannelHandler 就是 ChannelUpstreamHandler 或 ChannelDownstreamHandler，在 ChannelPipeline 的传递过程中，事件只会调用匹配流的 ChannelHandler。

ChannelUpstreamHandler 或 ChannelDownstreamHandler 在事件流的过滤器链中既可以终止整个流程，也可以通过调用 ChannelHandlerContext.sendUpstream(ChannelEvent) 或 ChannelHandlerContext.sendDownstream(ChannelEvent)，将当前的事件继续传递下去。Upstream Event 是被 UpstreamHandler 从下往上依次处理的，而 Downstream Event 是被 DownstreamHandler 从上往下依次处理的，这种上下关系实质上就是在初始化的时候，往 ChannelPipeline 里添加的 Handler 的先后顺序。也就是说，Upstream Event 用于请求外部请求的过程，而 Downstream Event 用于处理服务器向外发送请求的过程。

### 3.4.7 Netty 编程

由于 Netty 与 Mina 是同一人设计的，其设计思想大同小异，因此在开发过程中，Netty 和 Mina 一样简单，其良好的架构和设计思想，使开发者在开发过程中更容易进行网络应用程序的开发。下面讲解在 Java 中使用 Netty 的实例。



## 【范例】

服务端:

```
package com.hjc.demo.netty;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;
import io.netty.util.CharsetUtil;

public class NettyServer {
    public static int port;
    private static NioEventLoopGroup bossGroup = new NioEventLoopGroup();
    private static NioEventLoopGroup workGroup = new NioEventLoopGroup();

    public void initData() {
        port = 8300;
    }

    // Test Code
    public static void main(String[] args) {
        NettyServer server = new NettyServer();
        NettyServer.port = 8300;
        server.start();
        System.out.println("Netty 服务端已启动");
    }

    public void start() {
        ServerBootstrap bootstrap = new ServerBootstrap();
        bootstrap.group(bossGroup, workGroup);
        bootstrap.channel(NioServerSocketChannel.class);
        bootstrap.option(ChannelOption.SO_BACKLOG, 128);
```



```

// 通过 NoDelay 禁用 Nagle, 使消息立即发送出去, 不用达到一定的数据量时才发送出去
bootstrap.option(ChannelOption.TCP_NODELAY, true);
bootstrap.option(ChannelOption.SO_REUSEADDR, true);
// 保持长连接状态
bootstrap.childOption(ChannelOption.SO_KEEPALIVE, true);
bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        // 添加 String 编解码器
        pipeline.addLast(new StringDecoder(CharsetUtil.UTF_8));
        pipeline.addLast(new StringEncoder(CharsetUtil.UTF_8));
        // 业务逻辑处理
        pipeline.addLast(new NettyServerHandler());
    }
});
// 启动端口
ChannelFuture future;
try {
    future = bootstrap.bind(port).sync();
    if (future.isSuccess()) {
        System.out.println("端口" + port + "已绑定");
    }
} catch (InterruptedException e) {
    System.out.println("端口" + port + "已绑定");
}
}

public static void shut() {
    workGroup.shutdownGracefully();
    bossGroup.shutdownGracefully();
    System.out.println("端口" + port + "已解绑");
}
}

```

Netty 服务端的消息处理类:

```

package com.hjc.demo.netty;

import io.netty.channel.ChannelHandlerAdapter;
import io.netty.channel.ChannelHandlerContext;

```

```
import io.netty.channel.ChannelPromise;

public class NettyServerHandler extends ChannelHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        System.out.println("服务端收到消息: " + msg.toString());
        ctx.writeAndFlush("服务端收到了: " + msg.toString());
        ctx.writeAndFlush("msg from server");
        ctx.writeAndFlush("quit");
    }

    @Override
    public void write(ChannelHandlerContext ctx, Object msg,
        ChannelPromise promise) throws Exception {
        System.out.println("服务端发送消息: " + msg.toString());
    }

    @Override
    public void disconnect(ChannelHandlerContext ctx, ChannelPromise promise)
        throws Exception {
        System.out.println("客户端断开连接");
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        // System.out.println("channelActive");
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
        throws Exception {
        // System.out.println("exceptionCaught");
    }
}
```

客户端:

```
package com.hjc.demo.netty;
```

```

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;
import io.netty.util.CharsetUtil;

import java.net.InetSocketAddress;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class NettyClient {
    private ScheduledExecutorService executor = Executors
        .newScheduledThreadPool(1);
    public static final int PORT = 8300;
    public static final String IP = "localhost";

    public static EventLoopGroup group = new NioEventLoopGroup();

    public void connect(int port, String host, final int index)
        throws Exception {
        // 配置客户端 NIO 线程组
        try {
            Bootstrap b = new Bootstrap();
            b.group(group).channel(NioSocketChannel.class)
                .option(ChannelOption.TCP_NODELAY, true)
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    public void initChannel(SocketChannel ch)
                        throws Exception {
                        ChannelPipeline pipeline = ch.pipeline();
                        // 添加 String 编解码器
                        pipeline.addLast(new StringDecoder(

```



```

        CharsetUtil.UTF_8));
        pipeline.addLast(new StringEncoder(
            CharsetUtil.UTF_8));
        // 业务逻辑处理
        pipeline.addLast(new NettyClientHandler(index));
    }
});
// 发起异步连接操作
ChannelFuture future = b.connect(new InetSocketAddress(host, port));
future.channel().closeFuture().sync();
} finally {
    // 所有资源释放完毕之后, 清空资源, 再次发起重连操作
    executor.execute(new Runnable() {
        @Override
        public void run() {
            try {
                TimeUnit.SECONDS.sleep(1);
                try {
                    // 以下代码可以在 channel 断开的时候自动重连
                    // connect(PORT, IP, index); // 发起重连操作
                } catch (Exception e) {
                    e.printStackTrace();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
}

/**
 * @param args
 * @throws Exception
 */
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 3; i++) {
        new NettyClient().connect(PORT, IP, (i + 1));
    }
}
}

```

## Netty 客户端的消息处理类:

```
package com.hjc.demo.netty;

import io.netty.channel.ChannelHandlerAdapter;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelPromise;

public class NettyClientHandler extends ChannelHandlerAdapter {

    public int index = 0;

    public NettyClientHandler(int index) {
        this.index = index;
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        System.out.println("客户端" + index + "收到消息: " + msg.toString());
        if (msg.toString().equals("quit")) {
            ctx.channel().close();
            // NettyClient.group.shutdownGracefully();
        } else {
            ctx.writeAndFlush("msg from client");
        }
    }

    @Override
    public void write(ChannelHandlerContext ctx, Object msg,
        ChannelPromise promise) throws Exception {
        System.out.println("客户端" + index + "发送消息: " + msg.toString());
    }

    @Override
    public void disconnect(ChannelHandlerContext ctx, ChannelPromise promise)
        throws Exception {
        System.out.println("客户端" + index + "断开连接");
    }

    @Override
```



```

public void channelActive(ChannelHandlerContext ctx) throws Exception {
    System.out.println("客户端" + index + "连接上了");
    ctx.writeAndFlush("客户端发送的消息，服务端接收到了吗?");
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
    throws Exception {
    System.out.println("exceptionCaught");
}
}

```

## 【运行结果】

服务端：

端口 8300 已绑定

Netty 服务端已启动

服务端收到消息：客户端发送的消息，服务端接收到了吗？

服务端收到消息：msg from client

服务端收到消息：msg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from client

服务端收到消息：msg from client

服务端收到消息：客户端发送的消息，服务端接收到了吗？

服务端收到消息：msg from client

服务端收到消息：msg from client

服务端收到消息：客户端发送的消息，服务端接收到了吗？

服务端收到消息：msg from client

服务端收到消息：msg from client

服务端收到消息：msg from client

服务端收到消息：msg from clientmsg from client

服务端收到消息：msg from client

服务端收到消息: msg from client

服务端收到消息: msg from clientmsg from client

服务端收到消息: msg from client

服务端收到消息: msg from clientmsg from client

服务端收到消息: msg from client

服务端收到消息: msg from clientmsg from client

客户端:

客户端 1 连接上了

客户端 1 收到消息: 服务端收到了: 客户端发送的消息, 服务端接收到了吗? msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from clientmsg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from clientmsg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from clientmsg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from clientmsg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from clientmsg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from clientmsg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from clientmsg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from clientmsg from client

客户端 1 收到消息: msg from serverquit

客户端 1 收到消息: 服务端收到了: msg from clientmsg from clientmsg from server

客户端 1 收到消息: quit

客户端 2 连接上了

客户端 2 收到消息: 服务端收到了: 客户端发送的消息, 服务端接收到了吗?

客户端 2 收到消息: msg from serverquit

客户端 2 收到消息: 服务端收到了: msg from clientmsg from server

客户端 2 收到消息: quit

客户端 3 连接上了

客户端 3 收到消息: 服务端收到了: 客户端发送的消息, 服务端接收到了吗?

客户端 3 收到消息: msg from serverquit

客户端 3 收到消息: 服务端收到了: msg from clientmsg from serverquit

```

客户端 3 收到消息: 服务端收到了: msg from clientmsg from server
客户端 3 收到消息: quit 服务端收到了: msg from client
客户端 3 收到消息: msg from serverquit
客户端 3 收到消息: 服务端收到了: msg from clientmsg from clientmsg from server
客户端 3 收到消息: quit 服务端收到了: msg from client
客户端 3 收到消息: msg from serverquit
客户端 3 收到消息: 服务端收到了: msg from clientmsg from serverquit
客户端 3 收到消息: 服务端收到了: msg from clientmsg from clientmsg from server
客户端 3 收到消息: quit 服务端收到了: msg from client
客户端 3 收到消息: msg from serverquit
客户端 3 收到消息: 服务端收到了: msg from clientmsg from clientmsg from serverquit
客户端 3 收到消息: 服务端收到了: msg from clientmsg from server
客户端 3 收到消息: quit

```

**【代码解析】**Netty 的开发同样简单高效。创建 Netty 的服务端，首先需要创建一个 boss 线程组和一个 work 线程组。boss 线程组用于接收客户端的连接，而 work 线程组用于连接消息的处理，然后创建 ServerBootstrap 来绑定线程组，绑定 ServerChannel，设置 option 和绑定 ChannelHandler 之后，再调用 bind 方法绑定端口，即完成服务端的创建，ChannelHandler 中根据需求重写父类方法来实现业务逻辑。客户端的实现则通过 Bootstrap 类绑定 ServerChannel，设置 option 和绑定 ChannelHandler 之后调用 connect 方法与服务端建立连接。在 Netty 中通过调用 Channel 的 write 方法可发送消息。

## 3.5

## 总结

本章总结了 Java 网络编程相关知识，首先介绍了 TCP、UDP、HTTP、Socket、WebSocket 等协议的理论和特点，及其 Java 代码的实现。然后介绍了 BIO、NIO、AIO 等网络通信模型，并通过代码展现了其原理。最后分别介绍了 Mina 和 Netty 两款网络通信框架，它们由同一人设计，在底层原理、架构上都是惊人的相似的。本章只是蜻蜓点水般地介绍了 Mina 和 Netty 的主要特征，其实探究 Mina 和 Netty 是非常有必要的。因为现在很多的分布式框架，如阿里 Dubbo 的底层，就是基于高效稳定的 Netty 实现的。

# 第4章

## 数据交互

网络层建立起客户端与服务器的连接之后，就可以传输数据。传输数据时，为了让双方都能辨别，需要拟定一个双方都能解析的数据格式，就好比两个人要交流，一定要用双方都能明白的语言一样。这些在传输层中服务端与客户端相互交互的数据，就是本章要介绍的内容。

### 4.1

#### 数据传输格式

建立网络连接之后，客户端与服务器就可以进行数据传输，从物理本质上来说，网络传输其实是通过一系列的线路（光纤、双绞线等），经过电路调整变化，依据网络传输协议进行通信的过程。数据交换有多种格式，对于客户端和服务器来说，数据传输通常要考虑以下几点。

##### 1. 网络数据大小

网络数据的大小直接影响了带宽的占用，从而影响传输效率，网络流量很小的时候可能不会有太大的影响，但当服务器承受高并发的数据传输时，网络流量会达到一个峰值，若峰值太大很可能导致带宽占满，从而影响服务器的使用效率和用户体验。因此数据传输过程中，应尽量减少冗余数据，以节省带宽，提高传输效率。



## 2. 网络数据安全性

客户端与服务器交互的绝大部分数据是希望对外界隐藏的，这些数据很可能会涉及敏感数据，因此在网络传输的数据格式选择中，信息安全也是一个很重要的考虑因素。

## 3. 实现复杂度

数据传输过程中，需要对信息进行序列化和反序列化，因此序列化与反序列化的复杂度及效率也是影响数据传输效率的一个重要因素。而且，在实际项目中，为了适应需求的变更，也需要注意数据结构的可扩展性和可维护性。

## 4. 协议通用性

通常情况下，客户端与服务端都是不同的运行环境，因此通常需要传输的数据具有跨平台的优势，能够实现同步平台之间的跨平台通信，而不局限于同种平台之间的数据传输。

一般，数据传输格式需要满足以上几个条件，才能应对普遍需求，下面就对几种类别的数据类型进行介绍。

### 1. 自定义二进制 (Binary)

最简单直接的数据传输格式就是通过二进制进行传输，这样的信息体积非常小，但易读性和易写性都不尽如人意。自定义二进制传输需要服务器和客户端自己定义消息格式，并自己实现序列化和反序列化的方法和容错处理等，可扩展性也不强，很难得到广泛的使用，但它的自由度是很高的。如果开发团队有很强的技术水平，也可以自己开发一套传输协议。

### 2. 开源协议

最常见的开源协议库有 JSON、XML、Google 的 Protobuffer 及 Facebook 的 Thrift。这些开发库基本都提供了序列化和反序列化的库，并且在扩展性和容错处理方面都是非常不错的。Protobuffer 和 Thrift 是比 JSON、XML、Binary 体积更小、效率更高、使用更方便、支持语言更多、更高级的开源协议库。

### 3. 文本化协议

数据传输也可以直接使用文本格式，按照一定的协议标准组织结构进行传输，比如



常见的 JSON、XML，通过使用常见的开源库，它们也可以很方便地进行序列化和反序列化。由于是文本传输，因此数据的大小要大于二进制文件很多。但它的呈现形式却更接近人类语言，因此其可视化会更好、更容易维护、更方便调试。

常见的数据格式的特点如表 4-1 所示。

表 4-1 常见数据格式的特点

	XML	JSON	Protobuffer
数据结构支持	复杂结构	简单结构	复杂结构
数据保存方式	文本	文本	二进制
数据保存大小	大	一般	小
解析效率	慢	一般	快
语言支持程度	非常多	多	C++/Java/Python
开发难度	低	低	低
适用范围	数据交换	数据交换	数据交换

## 4.2 JSON 的使用及解析

JSON（JavaScript Object Notation，JavaScript 对象）是一种轻量级数据交换格式，是基于 ECMAScript 的一个子集，也是基于 JavaScript 对象表示语法的子集。但它是完全独立于语言的一种文本格式，使用了类 C 语言家族（包括 C、C++、C#、Java、JavaScript、Perl、Python 等）的习惯，对众多语言的友好使它成为数据交换的理想语言。JSON 具有很好的描述性与层级结构。

### 4.2.1 JSON 语法

JSON 语法主要有以下几个特点。

- （1）数据都是以键值对形式出现的，键与值之间以冒号分隔。
- （2）数据之间以逗号分隔。
- （3）大括号保存 JSON 对象。
- （4）中括号保存 JSON 数组。

## 4.2.2 JSON 对象

JSON 对象在大括号中书写，对象可以包含多个键值对，例如以下语句：

```
{
  "obj1":{
    "param11":"value1",
    "param12":2
  },
  "obj2":{
    "param21":"value1",
    "param22":2
  }
}
```

## 4.2.3 JSON 数组

JSON 数组在中括号中书写，数组可以包含多个对象，例如以下语句：

```
[
  "arrayvalue1",
  2,
  [
    [
      1,
      2
    ],
    [
      3,
      "4"
    ]
  ],
  {
    "param11":"value1",
    "param12":2
  },
  {
    "param21":"value1",
    "param22":2
  }
]
```

```
}
]
```

## 4.2.4 Java 中的 JSON 解析

以上都是对 JSON 语法的解释,那么在 Java 中 JSON 又是如何使用和解析的呢? JSON 的官网 <http://www.json.org> 中,给出了很多 JSON 构造和解析工具,使用最多的是 org.json 和 Json-lib。很多开源库也有 JSON 解析工具,如 Google 的 Gson、阿里的 Fastjson 等。

下面介绍几种 Json 类库的使用方法。

### 1. Json-lib

#### 【范例】

```
package com.hjc.demo;

import java.util.HashMap;
import java.util.Map;

import net.sf.json.JSONObject;

public class JsonlibDemo {

    public static void main(String[] args) {
        // Java 序列化为 Json
        JSONObject obj = new JSONObject();
        obj.setId(11);
        obj.setOrderId("2016_3_27");
        JsonSubObject subObject = new JsonSubObject();
        subObject.setCode(1001);
        subObject.setMsg("message");
        obj.setSubObject(subObject);
        JSONObject jsonObject = JSONObject.fromObject(obj);
        System.out.println("json string:");
        System.out.println(jsonObject.toString());
        // Json 反序列化为 Java
        // 如果对象中含有复杂子对象,如 List、Map 或自定义 JavaBean,需要使用以下的
```

classMap, 否则转换过程中会报错

```
Map<String, Class> classMap = new HashMap<String, Class>();
classMap.put("subObject", JsonSubObject.class);
obj = (JsonObject) JSONObject.toBean(jsonObject, JsonObject.class,
    classMap);
System.out.println("Java Object:");
System.out.println(obj);
System.out.println("id:" + obj.getId());
System.out.println("orderId:" + obj.getOrderId());
System.out.println("sub code:" + obj.getSubObject().getCode());
System.out.println("sub msg:" + obj.getSubObject().getMsg());
}
```

### 【运行结果】

```
json string:
{"id":1,"orderId":"2016_3_27","subObject":{"code":1001,"msg":"message"}}
Java Object:
com.hjc.demo.JsonObject@7b895df8
id:1
orderId:2016_3_27
sub code:1001
sub msg:message
```

**【代码解析】**使用 Json-lib 需要导入 net.sf.json.\*包, 使用 JSONObject 的 toBean 方法可将 JSONObject 对象转换为 JavaBean 对象, 而 JSONObject 的 fromObject 对象可将 JavaBean 对象转换为 JSONObject 对象。需要注意的是, 如果对象中含有复杂子对象, 如 List、Map 或自定义 JavaBean, 需要使用上述 classMap, 否则转换过程中会报错。

## 2. Gson

### 【范例】

```
package com.hjc.demo;

import com.google.gson.Gson;
```



```

public class GsonDemo {

    public static void main(String[] args) {
        String jsonString = null;
        // Java 序列化为 Json
        JsonObject obj = new JsonObject();
        obj.setId(11);
        obj.setOrderId("2016_3_27");
        JsonSubObject subObject = new JsonSubObject();
        subObject.setCode(1001);
        subObject.setMsg("message");
        obj.setSubObject(subObject);
        Gson gson = new Gson();
        jsonString = gson.toJson(obj);
        System.out.println("json string:");
        System.out.println(jsonString);
        // Json 反序列化为 Java
        obj = gson.fromJson(jsonString, obj.getClass());
        System.out.println("Java Object:");
        System.out.println(obj);
        System.out.println("id:" + obj.getId());
        System.out.println("orderId:" + obj.getOrderId());
        System.out.println("sub code:" + obj.getSubObject().getCode());
        System.out.println("sub msg:" + obj.getSubObject().getMsg());
    }
}

```

### 【运行结果】

```

json string:
{"id":1,"orderId":"2016_3_27","subObject":{"code":1001,"msg":"message"}}
Java Object:
com.hjc.demo.JsonObject@60bb6b37
id:1
orderId:2016_3_27
sub code:1001
sub msg:message

```

**【代码解析】**创建 Gson 对象后, 可使用 toJson 方法将 JavaBean 转换为 Json 字符串, 也可以使用 fromJson 方法将 Json 字符串转换为 JavaBean 对象。



### 3. Jackson

#### 【范例】

```
package com.hjc.demo;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.map.ObjectMapper;

public class JacksonDemo {

    public static void main(String[] args) {
        String jsonString = null;
        // Java 序列化为 Json
        try {
            Map jsonMap = new HashMap();
            ObjectMapper mapper = new ObjectMapper();
            jsonMap.put("param1", "value1");
            jsonMap.put("param2", "value2");
            JsonObject obj = new JsonObject();
            obj.setId(11);
            obj.setOrderId("2016_3_27");
            JsonSubObject subObject = new JsonSubObject();
            subObject.setCode(1001);
            subObject.setMsg("message");
            obj.setSubObject(subObject);
            jsonMap.put("param3", mapper.writeValueAsString(obj));
            jsonString = mapper.writeValueAsString(jsonMap);
            System.out.println("Json String:");
            System.out.println(jsonString);
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Json 反序列化为 Java
        try {
            ObjectMapper mapper2 = new ObjectMapper();
            JsonNode root = mapper2.readTree(jsonString);
```

```

        Map jsonMap = mapper2.readValue(jsonString, Map.class);
        System.out.println("Java Object:");
        System.out.println(jsonMap);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

### 【运行结果】

```

Json String:
{"param1":"value1","param2":"value2","param3":{"id":1,"orderId":"2016_3_27","subObject":{"code":1001,"msg":"message"}}}
Java Object:
{param1=value1, param2=value2, param3={id=1, orderId="2016_3_27", "subObject": {"code":1001, "msg":"message"}}}

```

**【代码解析】**创建 ObjectMapper 类的对象，通过调用 writeValueAsString 可将 JavaBean 转换为 Json 字符串，调用 readValue 或 readTree 可将 JavaBean 读取到 JsonNode 或 Map 中。

## 4. Fastjson

### 【范例】

```

package com.hjc.demo;

import com.alibaba.fastjson.JSON;

public class FastjsonDemo {

    public static void main(String[] args) {
        String jsonString = null;
        // Java 序列化为 Json
        JsonObject obj = new JsonObject();
        obj.setId(11);
        obj.setOrderId("2016_3_27");
        JsonSubObject subObject = new JsonSubObject();
        subObject.setCode(1001);
    }
}

```

```

subObject.setMsg("message");
obj.setSubObject(subObject);
jsonString = JSON.toJSONString(obj);
System.out.println("json string:");
System.out.println(jsonString);
// Json 反序列化为 Java
obj = JSON.parseObject(jsonString, JsonObject.class);
System.out.println("Java Object:");
System.out.println(obj);
System.out.println("id:" + obj.getId());
System.out.println("orderId:" + obj.getOrderId());
System.out.println("sub code:" + obj.getSubObject().getCode());
System.out.println("sub msg:" + obj.getSubObject().getMsg());
}
}

```

### 【运行结果】

```

json string:
{"id":1,"orderId":"2016_3_27","subObject":{"code":1001,"msg":"message"}}
Java Object:
com.hjc.demo.JsonObject@7b9d142f
id:1
orderId:2016_3_27
sub code:1001
sub msg:message

```

**【代码解析】**Fastjson 可通过使用 com.alibaba.fastjson.JSON 类来实现对 Json 的操作，通过调用 JSON 的 toJSONString 可以将 JavaBean 转换为 Json 字符串，调用 parseObject 可将 Json 字符串转换为 JavaBean。

以上几款开源的 JSON 工具库的性能也并不是完全一样的，其速度关系是：Fastjson>Gson>Jackson>Json-lib，这是目前最新版本的性能的比较。

## 4.3

### XML 的使用及解析

XML (Extensible Markup Language, 可扩展标记语言) 是常用的数据交互格式，也

是能对具有结构性的文件进行标记的语言。1998年2月，W3C正式批准了可扩展标记语言的标准定义，可扩展标记语言可以对文档和数据进行结构化处理，从而使各个组件之间进行交换，实现动态内容生成、企业集成和应用开发。可扩展标记语言可以使我們更准确地搜索、更方便地传送数据、更好地描述一些事物、传输或存储数据。

### 4.3.1 XML 的特征

XML 主要具备以下几个特征：

- (1) 可标记扩展语言。
- (2) 可标记性语言类似于超文本标记语言。
- (3) 设计宗旨是传输数据。
- (4) 没有预定义标签，需要自定义标签。
- (5) 具有自我描述性。
- (6) 是 W3C 推荐的标准。

### 4.3.2 数据共享

XML 的数据通过纯文本的形式进行存储，它不同于软件或硬件的存储方式，是一种独立的存储方式，XML 使共享数据更容易。

### 4.3.3 数据传输

使用 XML 可以很容易地在不同的系统之间交换数据。对开发人员来说，因不同系统之间存在差异，所以进行数据交互是最费时费力的事情。XML 可以通过各种不兼容的应用程序读取数据，以降低这种复杂性。

### 4.3.4 平台兼容

升级到新的系统通常需要转换很多数据，这个过程是非常耗时的，并且经常会丢失



不兼容的数据。XML 数据以文本格式存储，可以在不损失数据的情况下，更容易扩展或升级到新操作系统、新应用程序或新浏览器。

### 4.3.5 JSON 与 XML 的比较

#### 1. 可读性

JSON 的可读性与 XML 差不多，二者几乎是不相上下的。无论是 JSON 的简易语法，还是 XML 的规范标签，都有很好的可读性。

#### 2. 可扩展性

XML 在可扩展性上较 JSON 更胜一筹。

#### 3. 编码难度

XML 有很多编码工具，如 Dom4j、JDom 等，JSON 也有不少编码工具。在无工具的情况下，XML 会有更多的字符解析，JSON 也能很好地进行解析，但 JSON 更胜一筹。

#### 4. 解码难度

XML 的解析可以通过文档模型进行，即通过父标签引出一组标记，或通过节点的遍历来解析。而对于 JSON 的解析，如果已经知道数据结构就是非常简单的，但若不知道将是一件非常困难的事情。

#### 5. JSON 是否能取代 XML

答案当然是否定的。首先，XML 有更好的结构描述能力，在对数据结构有严格要求的场景下，XML 仍然是不二之选。其次，XML 具有更好的通用性，服务端与客户端之间的通信通用的是 XML，服务端有时也不能很好地对 JSON 进行编解码。

### 4.3.6 Java 中的 XML 解析

在 Java 中，解析 XML 要比解析 JSON 复杂一点，不过也有很多第三方支持的 XML 解析库，选用最合适的 XML 解析方式，可以提高解析效率。下面我们就通过 Java 代码来实现对 XML 的解析。



## 1. JDK 自带 Marshaller 与 Unmarshaller

### 【范例】

```
package com.hjc.demo.xml;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.StringReader;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;

/**
 * @author hjc 使用 JDK 自带类 Marshaller 与 Unmarshaller, 使客户端应用程序将 XML
 * 数据转换为 Java 内容对象树。Marshaller 类能使客户端应用程序将 Java 内容树转换回 XML
 * 数据
 *
 */
public class XmlJDKDemo {

    public static void main(String[] args) {
        XmlJDKDemo demo = new XmlJDKDemo();
        try {
            demo.run();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void run() throws IOException {
        StringBuffer xmlBuffer = new StringBuffer();
        // Javabean to Xml
        try {
            RootElement root = new RootElement();
            root.setVal1(100000000000000000001);
```

```

root.setVal2(1.00000000000000001d);
SubElement sub = new SubElement();
sub.setSubval1(1000);
sub.setSubval2("aaaaaaaaaaaa");
root.setVal3(sub);
JAXBContext context = JAXBContext.newInstance(RootElement.class);
Marshaller marshaller = context.createMarshaller();
File xmlFile = new File("D://marshaller.xml");
marshaller.marshal(root, xmlFile);
BufferedReader reader = new BufferedReader(new InputStreamReader(
    new FileInputStream(xmlFile)));
String line = null;
while ((line = reader.readLine()) != null) {
    xmlBuffer.append(line);
}
reader.close();
System.out.println(xmlBuffer.toString());
} catch (JAXBException e) {
    e.printStackTrace();
}
System.out.println("=====");
// Xml to Javabean
try {
    JAXBContext context = JAXBContext.newInstance(RootElement.class);
    Unmarshaller unmarshaller = context.createUnmarshaller();
    RootElement root = (RootElement) unmarshaller
        .unmarshal(new StringReader(xmlBuffer.toString()));
    System.out.println(root.getVal1());
    System.out.println(root.getVal2());
    System.out.println(root.getVal3().getSubval1());
    System.out.println(root.getVal3().getSubval2());
} catch (JAXBException e) {
    e.printStackTrace();
}
}
}

```

### Xml 根元素:

```
package com.hjc.demo.xml;
```

```
import javax.xml.bind.annotation.XmlRootElement;
```

```
/**
```

```
 * @author hjc Unmarshaller 的文档根元素
```

```
 *
```

```
 */
```

```
@XmlRootElement
```

```
public class RootElement {
```

```
    private long val1;
```

```
    private double val2;
```

```
    private SubElement val3;
```

```
    public RootElement() {
```

```
        super();
```

```
    }
```

```
    public long getVal1() {
```

```
        return val1;
```

```
    }
```

```
    public void setVal1(long val1) {
```

```
        this.val1 = val1;
```

```
    }
```

```
    public double getVal2() {
```

```
        return val2;
```

```
    }
```

```
    public void setVal2(double val2) {
```

```
        this.val2 = val2;
```

```
    }
```

```
    public SubElement getVal3() {
```

```
        return val3;
```

```
    }
```

```
    public void setVal3(SubElement val3) {
```

```
        this.val3 = val3;
```

```
    }
```

```
}
```



## Xml 子元素:

```

package com.hjc.demo.xml;

public class SubElement {
    private int subval1;
    private String subval2;
    public SubElement(){
        super();
    }

    public int getSubval1() {
        return subval1;
    }

    public void setSubval1(int subval1) {
        this.subval1 = subval1;
    }

    public String getSubval2() {
        return subval2;
    }

    public void setSubval2(String subval2) {
        this.subval2 = subval2;
    }
}

```

## 【运行结果】

```

<?xml version="1.0" encoding="UTF-8"
standalone="yes"?><rootElement><val1> 10000000000000000000</val1><val2>1.0
</val2><val3><subval1>1000</subval1><subval2>aaaaaaaaaa</subval2></val
3></rootElement>

=====
10000000000000000000
1.0
1000
aaaaaaaaaa

```

【代码解析】使用 JDK 自带类 Marshaller 与 Unmarshaller，使客户端应用程序将

XML 数据转换为 Java 内容对象树。Marshaller 类也能将 Java 内容对象树转换回 XML 数据。解析的 XML 根元素类需要加上注解@XmlRootElement。

## 2. DOM（基于 XML 文档树结构的解析）

### 【范例】

Xml 数据：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<RootElement>
<val1>1000000000000000</val1>
<val2>1.0000001</val2>
<val3>
<SubElement>
<subval1>1000</subval1>
<subval2>aaaaaaaa</subval2>
</SubElement>
</val3>
</RootElement>
```

Xml Dom 操作：

```
package com.hjc.demo.xml;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
```



```
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

/**
 * @author hjc dom 解析 XML 文件
 *
 */
public class DomDemo {
    public Document document;
    public String fileName;

    public static void main(String[] args) {
        DomDemo demo = new DomDemo();
        demo.init();
        demo.fileName = "D:\\dom.xml";
        demo.createXml(demo.fileName);
        demo.parserXml(demo.fileName);
    }

    public void init() {
        try {
            DocumentBuilderFactory factory = DocumentBuilderFactory
                .newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            this.document = builder.newDocument();
        } catch (ParserConfigurationException e) {
            System.out.println(e.getMessage());
        }
    }

    public void createXml(String fileName) {
        Element root = this.document.createElement("RootElement");
        this.document.appendChild(root);
        Element val1 = this.document.createElement("val1");
        val1.appendChild(this.document.createTextNode("1000000000000000"));
        root.appendChild(val1);
        Element val2 = this.document.createElement("val2");
        val2.appendChild(this.document.createTextNode("1.0000001"));
    }
}
```

```

root.appendChild(val2);
Element subElement = this.document.createElement("SubElement");
Element subval1 = this.document.createElement("subval1");
subval1.appendChild(this.document.createTextNode("1000"));
subElement.appendChild(subval1);
Element subval2 = this.document.createElement("subval2");
subval2.appendChild(this.document.createTextNode("aaaaaaa"));
subElement.appendChild(subval2);
Element val3 = this.document.createElement("val3");
val3.appendChild(subElement);
root.appendChild(val3);
TransformerFactory tf = TransformerFactory.newInstance();
try {
    Transformer transformer = tf.newTransformer();
    DOMSource source = new DOMSource(document);
    transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    PrintWriter pw = new PrintWriter(new FileOutputStream(fileName));
    StreamResult result = new StreamResult(pw);
    transformer.transform(source, result);
    System.out.println("生成 XML 文件成功!");
} catch (TransformerConfigurationException e) {
    System.out.println(e.getMessage());
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (TransformerException e) {
    System.out.println(e.getMessage());
}
}

public void parserXml(String fileName) {
    try {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document document = db.parse(fileName);
        NodeList employees = document.getChildNodes();
        for (int i = 0; i < employees.getLength(); i++) {
            Node employee = employees.item(i);

```

```

        NodeList employeeInfo = employee.getChildNodes();
        for (int j = 0; j < employeeInfo.getLength(); j++) {
            Node node = employeeInfo.item(j);
            NodeList employeeMeta = node.getChildNodes();
            for (int k = 0; k < employeeMeta.getLength(); k++) {
                System.out.println(employeeMeta.item(k).getNodeName()
                    + ":" + employeeMeta.item(k).getTextContent());
            }
        }
    }
    System.out.println("解析完毕");
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (ParserConfigurationException e) {
    System.out.println(e.getMessage());
} catch (SAXException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
}
}

```

### 【运行结果】

```

生成 XML 文件成功!
#text:1000000000000000
#text:1.0000001
#text:

SubElement:
1000
aaaaaaaaa

#text:

解析完毕

```

**【代码解析】**DOM 解析 XML 需要根据 Document 获取 XML 文档中所有的 Node，然后遍历 Node 读取 XML 文档。



### 3. SAX (基于事件流的解析)

#### 【范例】

```
package com.hjc.demo.xml;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SaxDemo {

    public static void main(String[] args) {
        SaxDemo demo = new SaxDemo();
        demo.parserXml("D://dom.xml");
    }

    public void createXml(String fileName) {
        System.out.println("<<" + fileName + ">>");
    }

    public void parserXml(String fileName) {
        SAXParserFactory saxfac = SAXParserFactory.newInstance();
        try {
            SAXParser saxparser = saxfac.newSAXParser();
            InputStream is = new FileInputStream(fileName);
            saxparser.parse(is, new MySAXHandler());
        } catch (ParserConfigurationException e) {
            e.printStackTrace();
        } catch (SAXException e) {
            e.printStackTrace();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```
e.printStackTrace();
    }
}

class MySAXHandler extends DefaultHandler {
    boolean hasAttribute = false;
    Attributes attributes = null;

    public void startDocument() throws SAXException {
        System.out.println("文档开始打印了");
    }

    public void endDocument() throws SAXException {
        System.out.println("文档打印结束了");
    }

    public void startElement(String uri, String localName, String qName,
        Attributes attributes) throws SAXException {
        if (qName.equals("rootElement")) {
            return;
        }
        System.out.println(qName + ":");
        if (attributes.getLength() > 0) {
            this.attributes = attributes;
            this.hasAttribute = true;
        }
    }

    public void endElement(String uri, String localName, String qName)
        throws SAXException {
    }

    public void characters(char[] ch, int start, int length)
        throws SAXException {
        if (length > 1) {
            System.out.println(new String(ch, start, length));
        }
    }
}
```

**【运行结果】**

```

文档开始打印了
RootElement:
val1:
1000000000000000
val2:
1.00000001
val3:
SubElement:
subval1:
1000
subval2:
aaaaaaaaa
文档打印结束了

```

**【代码解析】**通过 SAXParser 的 parse 方法解析从 XML 文件读取的 InputStream 流来读取，同时绑定一个继承自 DefaultHandler 的 XML 解析处理类，并在其中实现解析打印的方法。

**4. DOM4J**

DOM4J 是一款很优秀的 Java XML API，也是一个开放源代码的软件，具有性能优异、功能强大和极易使用的特点。现在，越来越多的 Java 软件都使用 DOM4J 读写 XML，特别值得一提的是，Sun 的 JAXM 也在使用 DOM4J。

**【范例】**

Dom4jDemo.java:

```

package com.hjc.demo.xml;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.util.Iterator;

import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.DocumentHelper;

```

```
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;

public class Dom4jDemo {

    public static void main(String[] args) {
        Dom4jDemo demo = new Dom4jDemo();
        demo.createXml("D://dom4j.xml");
        demo.parserXml("D://dom4j.xml");
    }

    public void createXml(String fileName) {
        Document document = DocumentHelper.createDocument();
        Element rootElement = document.addElement("rootElement");
        Element val1 = rootElement.addElement("val1");
        val1.setText("100000000000000");
        Element val2 = rootElement.addElement("val2");
        val2.setText("1.000000000000001");
        Element subElement = rootElement.addElement("subElement");
        Element subval1 = subElement.addElement("subval1");
        subval1.setText("1000");
        Element subval2 = subElement.addElement("subval2");
        subval2.setText("aaaaaaaaa");
        try {
            Writer fileWriter = new FileWriter(fileName);
            XMLWriter xmlWriter = new XMLWriter(fileWriter);
            xmlWriter.write(document);
            xmlWriter.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    public void parserXml(String fileName) {
        File inputXml = new File(fileName);
        SAXReader saxReader = new SAXReader();
        try {
            Document document = saxReader.read(inputXml);
            Element root = document.getRootElement();
        }
    }
}
```

```

        System.out.println(root.getName() + ":" + root.getText());
        for (Iterator i = root.elementIterator(); i.hasNext();) {
            Element val = (Element) i.next();
            System.out.println(val.getName() + ":" + val.getText());
            for (Iterator j = val.elementIterator(); j.hasNext();) {
                Element node = (Element) j.next();
                System.out.println(node.getName() + ":" + node.getText());
            }
        }
    } catch (DocumentException e) {
        System.out.println(e.getMessage());
    }
    System.out.println("dom4j parserXml");
}
}

```

### 【运行结果】

```

rootElement:
val1:1000000000000000
val2:1.0000000000000001
subElement:
subval1:1000
subval2:aaaaaaaaa
dom4j parserXml

```

**【代码解析】**DOM4J 通过 SAXReader 解析，类似于 DOM 解析。它也是通过 Document 对象获取所有的 Element 的，并通过 Element 对象获取节点的相关信息。

## 5. JDOM

JDOM 的出现减少了 DOM、SAX 的编码量，极大地减少了代码量。它适合功能简单，如解析、创建等的需求，但在底层 JDOM 还是使用 SAX（最常用）、DOM、Xanan 文档。

### 【范例】

```

package com.hjc.demo.xml;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;

```



```
import java.io.IOException;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;

/**
 *
 * JDOM 生成与解析 XML 文档
 *
 */
public class JDomDemo {

    public static void main(String[] args) {
        JDomDemo demo = new JDomDemo();
        demo.createXml("D://jdom.xml");
        demo.parserXml("D://jdom.xml");
    }

    public void createXml(String fileName) {
        Element root = new Element("rootElements");
        Document document = new Document(root);
        Element val1 = new Element("val1");
        val1.setText("100000000000000");
        root.addContent(val1);
        Element val2 = new Element("val2");
        val2.setText("1.000000000000001");
        root.addContent(val2);
        Element subElement = new Element("subElement");
        Element subval1 = new Element("subval1");
        subval1.setText("1000");
        subElement.addContent(subval1);
        Element subval2 = new Element("subval2");
        subval2.setText("aaaaaaaaaa");
        subElement.addContent(subval2);
        root.addContent(subElement);
        XMLOutputter XMLOut = new XMLOutputter();
```

```

try {
    XMLOut.output(document, new FileOutputStream(fileName));
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}

public void parserXml(String fileName) {
    SAXBuilder builder = new SAXBuilder(false);
    try {
        Document document = builder.build(fileName);
        Element root = document.getRootElement();
        List vals = root.getChildren();
        System.out.println((root).getName());
        for (int i = 0; i < vals.size(); i++) {
            Element val = (Element) vals.get(i);
            List subval = val.getChildren();
            if (subval.size() == 0) {
                System.out.println((val).getName() + ":" + val.getValue());
            }
            for (int j = 0; j < subval.size(); j++) {
                System.out.println(((Element) subval.get(j)).getName()
                    + ":" + ((Element) subval.get(j)).getValue());
            }
        }
    } catch (JDOMException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

### 【运行结果】

```

rootElements
val1:1000000000000000
val2:1.0000000000000001

```

```
subval1:1000  
subval2:aaaaaaaaaa
```

【代码解析】JDOM 也使用 Document 遍历所有的 Element 元素来获取具体节点上的信息。

虽然 XML 与 JSON 相比有很多不足之处，但在实际应用中，仍有很多场景是需要使用 XML 的，具体的数据传输格式的选择要根据具体的需求而定。

## 4.4

## Google Protocol Buffer 的介绍及使用

Protocol Buffer 是使用 Google 开发的一款开源数据交换格式，它独立于语言、独立于平台。Google 也提供多种语言的实现，每种语言都包含相应的编译器及库文件。Protobuffer（Protocol Buffer 的简称）采用二进制传输，比 JSON 及 XML 的体积都要小很多。但由于是二进制，所以它的可读性没有 JSON 和 XML 好。由于 Protobuffer 有很好的跨平台兼容性，所以它常用于分布式系统数据交换、网络传输、数据存储等。

### 4.4.1 Protobuffer 的安装与编译

Protobuffer 的官方网站是：<https://developers.google.com/protocol-buffers/>，开源地址是：<https://github.com/google/protobuf>，可以直接下载 Protobuffer 的源码，下载 `protobuf-2.5.0.tar.gz` 和 `protoc-2.5.0-win32.zip` 两个压缩包并分别解压。

在安装 Protobuffer 之前需要先安装 Maven。Maven 是一款项目管理工具，可以对项目进行包管理、编译、测试、发布等，这里就不详细介绍了。Maven 的官方下载地址是：<http://maven.apache.org/>，下载完成后解压到任意目录下，但需要把其 `bin` 目录添加到环境变量。这里需要利用 Maven 对 Protobuffer 进行编译。

将 `protoc-2.5.0-win32.zip` 中解压出来的 `protoc.exe` 移动到 `protobuf-2.5.0/src` 目录下，`protoc.exe` 文件是 Protobuffer 进行编译的可执行程序。进入 `protobuf-2.5.0/java`，运行 cmd 窗口，执行 `mvn package`，对 java 项目进行编译并打包 `proto-2.5.0.jar`，这里得到的 jar 就是我们在项目中需要引入并使用 Protobuffer 的 java 支持库。

此时就将 Protobuffer 编译好了，protoc.exe 就是需要将 proto 文件编译成 java 类的文件，protobuf-2.5.0.jar 就是开发中的 java 支持类库。

### 4.4.2 Protobuffer 的语法

#### 1. 标识符

Protobuffer 协议的标识符为 message 或 enum，message 代表消息类型，enum 代表枚举类型，在通过 Protobuffer 的编译器编译之后，它们都生成 Java 中的一个类。

#### 2. 修饰符

协议字段格式：role type name = tag [default value];

其中，role 有以下三种取值。

required: 该字段不能为空，必须传递值，否则 message 不能被正确初始化。

optional: 该字段可以为空，不管该字段是否传值，message 都能正确初始化。

repeated: 重复的字段，等同动态数组，编译成 Java 后即为 List，但是其数据可以为空。

#### 3. 数据类型

Protobuffer 中的数据类型与 Java 中的数据类型的对照关系如表 4-2 所示。

表 4-2 Protobuffer 中的数据类型与 Java 中的数据类型的对照表

Proto Type	Java Type
double	double
float	float
int32	int
int64	long
uint32	int
uint64	long
sint32	int
sint64	long
fixed32	int
fixed64	long
sfixed32	int



续表

Proto Type	Java Type
sfixed64	long
bool	boolean
string	string
bytes	bytestring

## 4. Package

在 Proto 文件中，可以用 Package 指定 Java 的包名，如：

```
package com.hjc.demo
```

该包名生成的 Java 文件的包名就是 com.hjc.demo。

## 5. Option

Option 可以定义一些常用选项，如以下几种。

java\_package: 指定 Java 文件的包名，并输出到指定目录下。

java\_outer\_classname: 指定 Java 文件的类型。

optimize\_for: 它的值为 SPEED（缺省条件，生成的代码效率高，代码占用空间大）、CODE\_SIZE（生成的代码占用空间小、效率低）或 LITE\_RUNTIME（生成的代码效率高，占用空间小，但其反射功能较弱）。

### 4.4.3 生成 Java 类

了解了 Protobuffer 的语法后，就可以开始写一个 proto 文件，然后利用 Protobuffer 的编译器编译为 Java 文件。编译 Java 文件需要使用 protoc.exe 编译器，使用 cmd 窗口进行编译，编译命令如下：

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR  
--python_out=DST_DIR path/to/file.proto
```

上述命令的参数意义如下：

(1) protoc 为 Protocol Buffer 提供的命令行编译工具。

(2) --proto\_path 等同于 -I 选项，指定待编译的 .proto 消息定义文件的目录。

(3) `--cpp_out` 表示生成 C++ 代码, `--java_out` 表示生成 Java 代码, `--python_out` 表示生成 Python 代码, 其后的目录为生成后的代码存放的目录。

(4) `path/to/file.proto` 表示待编译的消息定义文件。

#### 4.4.4 Eclipse 的 protobuf-dt 插件

其实, 使用 `protoc.exe` 编译的步骤可以交给 Eclipse 的 `protobuf-dt` 插件来完成, 安装 Eclipse 的地址是: <http://protobuf-dt.googlecode.com/git/update-site>, 如果安装速度太慢只能通过翻墙来实现, 这款插件还是非常实用的。

安装完成之后还需要对 `protobuf-dt` 插件进行一些设置, 选择 Eclipse 的 Preferences, 展开 Protocol Buffer, 再点击 Compiler, 勾选 `Compile .proto files on save`, 然后在 Main 的选项卡下选择 `protoc.exe` 编译器的位置 (见图 4-1)。

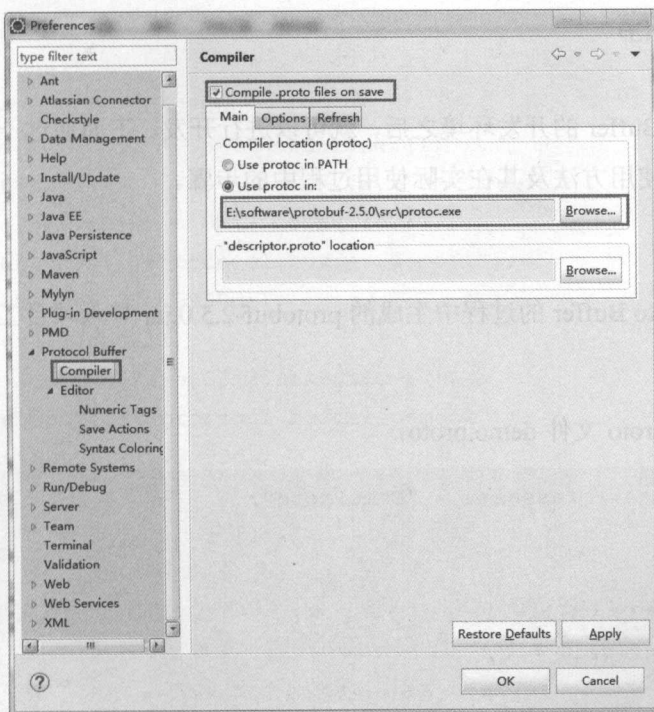


图 4-1 protobuf-dt 设置图

设置好 `protobuf-dt` 插件之后, 还需要设置 Java 文件的输出目录, 选择 Options 选项卡, 勾选 `Generate Java`, 然后指定 Java 文件输出的目录名字 (见图 4-2)。

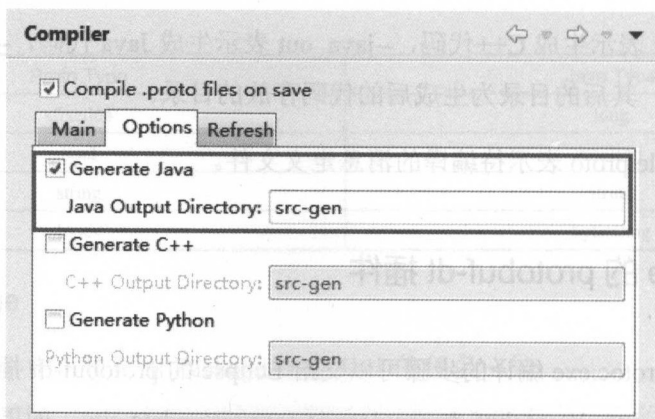


图 4-2 protobuf-dt 插件输出目录设置

设置完成后，在 Eclipse 中新建 proto 文件并编辑，每次保存之后，Eclipse 会自动将 proto 文件编译为 Java 文件，并放在设置的输出目录中。

#### 4.4.5 示例程序

配置好 ProtoBuffer 的开发环境之后，就可以进行开发。下面通过一个示例程序来讲解 ProtoBuffer 的使用方法及其在实际使用过程中的步骤。

##### 【案例】

先将编译 Proto Buffer 的过程中生成的 `protobuf-2.5.0.jar` 导入项目工程，并加入构建路径中。

给出指定的 proto 文件 `demo.proto`：

```
option java_outer_classname = "DemoProto";

message Obj1 {
    required int32 val1 = 1;
    repeated Obj2 val2 = 2;
}

message Obj2 {
    required string subval1 = 1;
    optional double subval2 = 2;
}
```

手动编译 Java 文件或 Eclipse 自动编译为 DemoProto.java（代码由 Proto Buffer 自动生成，文件太大，这里就不详细讲解），将此文件导入项目工程，此文件不能进行改动。

使用此 proto 文件的 PbDemo.java:

```
package com.hjc.demo.pb;

import com.hjc.demo.pb.DemoProto.Obj1;
import com.hjc.demo.pb.DemoProto.Obj2;

public class PbDemo {

    public static void main(String[] args) {
        PbDemo demo = new PbDemo();
        demo.run();
    }

    public void run() {
        // 构建 Protocol Javabean
        Obj1.Builder obj1 = Obj1.newBuilder();
        obj1.setVal1(1);
        Obj2.Builder obj21 = Obj2.newBuilder();
        obj21.setSubval1("protocol buffer demo 1");
        obj21.setSubval2(1.9999d);
        obj1.addVal2(obj21);
        Obj2.Builder obj22 = Obj2.newBuilder();
        obj22.setSubval1("protocol buffer demo 2");
        obj22.setSubval2(1.9999d);
        obj1.addVal2(obj22);
        System.out.println(obj1);
        System.out.println(obj1.build());
        // 接收到的 proto 直接对应 Protocol Javabean
        System.out.println("接收到的 proto 直接对应 Protocol Javabean");
        System.out.println("val1:" + obj1.getVal1());
        System.out.println("val2:" + obj1.getVal2List());
    }
}
```



**【运行结果】**

```
com.hjc.demo.pb.DemoProto$Obj1$Builder@39242445
```

```
val1: 1
```

```
val2 {
```

```
  subval1: "protocol buffer demo 1"
```

```
  subval2: 1.9999
```

```
}
```

```
val2 {
```

```
  subval1: "protocol buffer demo 2"
```

```
  subval2: 1.9999
```

```
}
```

接收到的 proto 直接对应 Protocol Javabean

```
val1:1
```

```
val2:[subval1: "protocol buffer demo 1"
```

```
subval2: 1.9999
```

```
, subval1: "protocol buffer demo 2"
```

```
subval2: 1.9999
```

```
]
```

**【代码解析】**可以看出，Proto Buffer 不仅数据体积小，使用起来也非常方便，只要双方协定好相同的 proto 文件，然后分别生成相应语言的类，即可通过 Builder 构建 Protobuf 对象，无论序列化还是反序列化，通过 Builder 都是很容易进行操作的。

以上便是在 Java 中使用 Proto Buffer 的简单示例，Proto Buffer 具有便捷、易扩展、易开发等特点，使其成为越来越热门的数据交换格式。Proto Buffer 提供了非常有用并且实用的功能，特别是针对序列化，如文件流、网络流等，也提供了完整的官网文档和规范的命名规则。

## 4.5

### 总结

本章介绍了在服务端与客户端通信时的数据交换格式，并分别介绍了 JSON、XML、Protocol Buffer 的特点及其在 Java 中的使用，同时也对它们做了各方面的对比。它们各有优势，也各有劣势，使用哪种数据格式需要根据业务场景进行选择。

## 第5章

# 数据缓存与持久化

玩家的游戏数据被传送到服务器时，有的进行处理后需要立即返回，有的需要临时存储在服务器，有的则需要缓存在服务器。本章将介绍在服务器中游戏数据的存储方式。游戏数据可以分为玩家数据和游戏数据，玩家数据是和玩家相关的信息的记录，随时都在变化；游戏数据是整个服务器公用的数据，基本上是不会变化的，如游戏中的资源数据（如游戏地图、怪物 AI）等。一般情况下，游戏数据服务器在启动时就会加载到内存中，而玩家数据会进行缓存或持久化。游戏数据一般由策划提供数据表（XML 或 CSV 表文件），然后由服务器读取，加载到内存中使用；玩家数据由服务器与客户端交互的玩家数据组成，有的需要缓存，有的则需要持久化。本章重点讲解玩家数据的存储。

### 5.1

## 游戏数据存储

在游戏服务器中，玩家的游戏数据基本上可以划分为“热数据”和“冷数据”两种。热数据即使用较频繁、生命周期较短的游戏数据；冷数据就是在交互中使用频率较低、生命周期较长的游戏数据。热数据，如玩家的位置信息、buff 状态、技能效果、剩余冷却时间等，适合使用缓存进行存储；冷数据，如玩家的等级、角色名、装备等，适合使用数据库进行持久化。

### 5.1.1 数据分类

在一个游戏服务器中，游戏的数据可以划分为几个不同的类型。按数据所属来分，分为全局数据和玩家个人数据；按玩家状态来分，可以分为在线玩家数据与离线玩家数据。

#### 1. 全局数据

游戏中的排行榜信息、联盟信息、国家信息、竞技场信息等全服所有玩家共享的信息，读取操作很频繁，所以可以把这类信息直接放进缓存中，这样玩家读取时就不用每次都到数据库读取，而是通过读取缓存来读取，减少数据的负载。当全局数据改变时，再通过实时或定期同步的方式同步到数据库。

#### 2. 在线玩家数据

服务器中的在线玩家数量多的时候能达到上千人，这时的数据量是巨大的，如果所有的数据读取操作都直接调用数据库，对数据库的压力也是非常大的，所以玩家读写最频繁部分的数据，必定要在缓存中进行操作；而玩家在游戏中读取不频繁的数据，如等级、角色名、装备等信息也可以适当设计到缓存中进行读写，可以采用主线程进行缓存操作，而数据库操作采用异步方式同步。具体的缓存方案设计还应结合实际情况，如在线人数、接口访问频率等因素。

#### 3. 离线玩家数据

离线玩家数据中，经常被其他在线玩家访问的数据可以被设计到缓存中进行访问，而一些不经常被访问的信息则持久化在数据库中，如果需要访问离线数据可通过异步方式操作。

### 5.1.2 数据缓存方式

游戏数据的缓存可以使用 Java 自身的内存（List、Map 等）或其他第三方的缓存中间件，比较常用的有 Memcache 和 Redis，它们都是内存数据库，所有的数据操作都在内存中，因此它们经常被用作第三方的缓存中间件。Memcache 和 Redis 均提供多种语言的支持，Memcache 在 Java 中有 spymemcached、xmemcached，Redis 在 Java 中有 spring-data-redis、jedis。

值得一提的是，Redis 和 Memcache 也是有一些区别的，虽然它们都是通过 key-value 方式来存储的，但 Memcache 存储的值仅支持 string 类型，而 Redis 支持 string、set、sorted set、hash、list 五种数据类型。Redis 还提供定期回写数据到本地的功能，实现数据的持久化，因此它也可用来做持久化的数据库，而 Memcache 并不提供这一功能。

5.1.3 数据持久化方式

游戏缓存的持久化可以使用关系型数据库 MySQL、NoSQL 数据库 Mongo 或带本地持久化功能的内存数据库 Redis。MySQL 的关系型数据库是大家比较熟悉的数据库类型，拥有索引、事务操作、表关联、完整性约束、标准 SQL 语句等最完整的数据库功能，它的使用以表为基本单位，而最近兴起的 NoSQL 数据库的概念也是很热门的，它灵活、便捷、易扩展及高效率的特点使其出现在越来越多的架构方案中。

最热门也最常用的当属 Mongo，它采用类 JSON 的 BSON 数据格式进行存储，支持丰富的数据表达，支持索引，查询语言非常丰富，也是最接近关系型数据库的 NoSQL 数据库，它的使用以文档为基本单位。

5.1.4 数据库的比较

下面对 MySQL、Redis、Memcache 及 Mongo 的一些特性进行一些比较（见表 5-1）。

表 5-1 数据库比较

数 据 库	类 型	数据存储	特 征
Redis	键值内存数据库	String、list、set、hash、sorted set	(1) 消息发布/订阅模式。 (2) 主从复制。 (3) 硬盘持久化
Memcached	键值内存数据库	key-value map	多线程
MySQL	关系型数据库	行列构成的数据表	(1) 支持事务（InnoDB 引擎）。 (2) 主从复制
MongoDB	NoSQL 文档型数据库	BSON 文档	(1) 支持 map-reduce 操作类型。 (2) 主从复制。 (3) 空间索引



## 5.2

## MySQL 的介绍及使用

MySQL 是一个开源的关系型数据库管理系统，又叫作 RDBMS。MySQL 的数据库系统使用最常用的数据库管理语言——SQL 语言（Structured Query Language），又叫作结构化查询语言。MySQL 属于关系型数据库，即在 MySQL 中，数据以表的形式存储，而一个数据库中存在着多张表，数据库中所有的表的关系，即为关系型。

### 5.2.1 特点

MySQL 由一家瑞典公司 MySQL AB 开发、运营并予以支持。它之所以非常流行，是因为其具备以下优点。

(1) MySQL 是基于开源许可发布的，并且无须支付任何费用即可免费使用。

(2) MySQL 本身的功能非常强大，虽然不复杂，却能够与绝大多数功能强大但价格昂贵的数据库软件匹敌。

(3) MySQL 使用的是业内最标准的 SQL 数据库语言。

(4) MySQL 可以运行在多个操作系统中，并且支持多种语言，包括 PHP、PERL、C、C++ 及 Java 等语言。

(5) MySQL 的操作十分迅速，即使面对大型数据集也没有压力。

(6) MySQL 是非常适合 PHP 等 Web 开发者使用的语言。

(7) MySQL 支持大型数据库，最高可在一个表中容纳 5 千多万行数据。虽然每张表的默认文件大小限制为 4GB，但如果操作系统支持，仍然可以将其理论限制增加到 800 万 TB。

(8) MySQL 可以进行自定义。基于开源 GPL 许可，开发者可以自由修改 MySQL，以便适应特殊的开发环境。

## 5.2.2 数据类型

MySQL 有三种数据类型，分别为数字、日期和时间、字符串，这三大类又可以细分出许多子类型。

### 1. 数字类型

整数：tinyint、smallint、mediumint、int、bigint。

浮点数：float、double、real、decimal。

### 2. 日期和时间

日期和时间：date、time、datetime、timestamp、year。

### 3. 字符串类型

字符串：char、varchar。

文本：tinytext、text、mediumtext、longtext。

二进制（可用来存储图片、音乐等）：tinyblob、blob、mediumblob、longblob。

## 5.2.3 MySQL 的使用

MySQL 采用 SQL（Structured Query Language，结构化查询语言）语法，因此标准 SQL 语法在 MySQL 中基本都适用，并且 MySQL 本身还提供了很多丰富的函数，如 DATE()、NOW()、SUM()等。

### 1. 登录 MySQL

安装完 MySQL 服务端与客户端之后，启动 MySQL 服务，然后通过客户端输入以下命令（也可通过图形工具操作）：

```
mysql -h 主机名 -u 用户名 -p
```

h：指定要登录的 MySQL 主机名，本地可用 127.0.0.1 或 localhost。

u：要登录的用户名。

p: 通知服务器将以密码形式登录，按回车键后输入密码。

登录成功后终端显示如下：

```
mysql>
```

## 2. 数据库操作

创建数据库命令如下：

```
create database 数据库名 [其他选项];
```

例如，创建一个名为 **demo** 的数据库，执行以下命令（utf8 为指定数据库的编码格式）：

```
create database demo character set utf8;
```

在 MySQL 中，对一个数据库进行操作之前需要选择数据库，否则会提示错误：

```
ERROR 1046(3D000): No database selected
```

指定数据库的方式有以下两种。

第一种：登录时指定。

```
mysql -D 数据库名
```

第二种：登录后用 **use** 指定。

```
use 数据库名
```

创建数据库表使用 **create table** 语句实现，如以下语句：

```
create table demo
(
    id int unsigned not null auto_increment primary key,
    name char(8) not null,
    sex char(4) not null,
    age tinyint unsigned not null,
    tel char(13) null default "-";
);
```

以上语句创建了一个包含 **id**、**name**、**sex**、**age** 和 **tel** 五个属性的数据库表。MySQL 的数据类型及事务、约束性等内容，本节不做过多介绍，感兴趣的读者可参考其他资料进行了解。

### 3. 插入数据

插入数据的语法格式如下：

```
insert [into] 表名 [(列名1, 列名2, 列名3, ...)] values (值1, 值2, 值3, ...);
```

其中，列名与值一一对应，如果前面不写列名，那么后面的值必须包含对应数据库中所有列的值，如以下语句：

```
insert into demo (name, sex, age) values("何金成", "男", 24);
```

以上语句表示插入一条名为“何金成”，性别为“男”，年龄为“24”的数据。

### 4. 查询数据

查询数据的语法格式如下：

```
select 列名称 from 表名称 [查询条件];
```

查询条件包括>、<、>=、<=、!=、in、not in、like、not like等众多条件，如以下语句：

```
mysql> select name, age from demo where age>=1 and name like '何%' or sex='男';
```

### 5. 更新表数据

更新表数据的语法格式如下：

```
update 表名称 set 列名称=新值 where 更新条件;
```

在使用 update 时，一定要注意 where 条件是否正确，若不正确可能更改到不需要更改的数据，造成不必要的麻烦，如以下语句：

```
update demo set name="何金成 2", age=18 where tel="13888888888";
```

### 6. 删除表数据

删除表数据的语法格式如下：

```
delete from 表名称 where 删除条件;
```

注意，同使用 update 时一样，在执行删除语句之前，一定要注意删除条件是否正确，避免造成不必要的人工误操作，如以下语句：

```
delete from demo where age<=20;
```



## 5.2.4 在 Java 中使用 MySQL

前面简单地讲解了 MySQL 的部分语法内容及数据库的简单使用，关系数据库涉及的内容很多，如 SQL 语法、事务、存储过程、触发器、完整性约束等，这里就不做过多讲述，想更深入了解 MySQL 的用户，可以翻阅相关书籍资料学习。本节重点讲解 MySQL 在 Java 中的使用方式。在 Java 中使用 MySQL 一般有 JDBC-ODBC Bridge、JDBC+Mysql 驱动，再往上封装的有 Hibernate、iBatis、MyBatis 几种方式，下面介绍最基本的 JDBC 实现与最常用的 ORM 框架 Hibernate 的实现。

### 1. JDBC

JDBC 是非常基础的数据库连接方式，主要有七个步骤：加载驱动、指定连接字符串及用户名密码等信息、建立数据库连接、获取数据库操作对象、执行 SQL 语句、处理结果及释放连接资源。下面通过代码进行讲解。

#### 【范例】

在 MySQL 中创建 t\_demo1 数据库：

```
create database t_demo1;
```

JDBCDemo.java:

```
package com.hjc.demo.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import com.hjc.demo.DemoBean;

public class JDBCDemo {
    public static void main(String[] args) {
        JDBCDemo demo = new JDBCDemo();
    }
}
```

```
demo.run();
}

public void run() {
    // JDBC 七步走
    Connection conn = null; // 数据库连接对象
    Statement stmt = null; // 数据库操作对象
    try {
        // 1. 加载驱动
        Class.forName("com.mysql.jdbc.Driver");
        System.out.println("数据库驱动加载成功");
        // 2. 指定连接字符串及用户名密码（你的密码不能这么简单哦！）
        String url = "jdbc:mysql://127.0.0.1:3306/demo";
        String user = "root";
        String pwd = "123456";
        // 3. 建立数据库连接
        conn = DriverManager.getConnection(url, user, pwd);
        System.out.println("数据库连接成功");
        // 4. 获取数据库操作对象 Statement
        stmt = conn.createStatement();
        // 5. 执行 SQL 语句
        // 创建表
        String sql = "create table t_demo1(id int(10),name varchar(50),sex varchar(4),createtime datetime)";
        int ret = stmt.executeUpdate(sql); // 执行 SQL 语句
        // 插入 3 条数据
        String insertSql = "insert into t_demo1 (id,name,sex,createtime)";
        values (1,'何金成 1','男','2016-4-7')";
        String insertSql2 = "insert into t_demo1 (id,name,sex,createtime)";
        values (2,'何金成 2','男','2016-4-7')";
        String insertSql3 = "insert into t_demo1 (id,name,sex,createtime)";
        values (3,'何金成 3','男','2016-4-7')";
        int ret21 = stmt.executeUpdate(insertSql);
        int ret22 = stmt.executeUpdate(insertSql2);
        int ret23 = stmt.executeUpdate(insertSql3);
        // 查询数据
        String querySql = "select * from t_demo1";
        ResultSet rs = stmt.executeQuery(querySql);
        // 6. 处理结果
        if (ret == 0) {
```

```
        System.out.println("建表 demo1 成功!");
    }
    if (ret21 != 0) {
        System.out.println("插入数据 1 成功!");
    }
    if (ret22 != 0) {
        System.out.println("插入数据 2 成功!");
    }
    if (ret23 != 0) {
        System.out.println("插入数据 3 成功!");
    }
    List<DemoBean> beans = new ArrayList<DemoBean>();
    System.out.println("查询数据库表数据");
    while (rs.next()) {
        // ResultSet 的遍历从下标 1 开始
        DemoBean bean = new DemoBean();
        int id = rs.getInt(1);
        String name = rs.getString(2);
        String sex = rs.getString(3);
        Date createtime = rs.getDate(4);
        bean.setId(id);
        bean.setName(name);
        bean.setSex(sex);
        bean.setCreatetime(createtime);
        beans.add(bean);
    }
    // print list
    for (DemoBean demoBean : beans) {
        System.out.println("=====");
        System.out.println(demoBean.getId());
        System.out.println(demoBean.getName());
        System.out.println(demoBean.getSex());
        System.out.println(demoBean.getCreatetime().toLocaleString());
    }
} catch (Exception e) {
    e.printStackTrace();
} finally {
    // 7.关闭对象, 释放资源
    if (stmt != null) {
        try {
```

```

        stmt.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
if (conn != null) {
    try {
        if (!conn.isClosed()) {
            conn.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
}
}

```

### 【运行结果】

数据库驱动加载成功

数据库连接成功

建表 demo1 成功!

插入数据 1 成功!

插入数据 2 成功!

插入数据 3 成功!

查询数据库表数据

```

=====
1
何金成 1
男
2016-4-7 0:00:00

```

```

=====
2
何金成 2
男
2016-4-7 0:00:00

```

```

=====
3
何金成 3
男
2016-4-7 0:00:00

```



【代码解析】使用 JDBC 只要记住以下七个步骤即可：使用 `Class.forName` 或 `DriverManager.registerDriver` 方法加载驱动；指定数据库连接字符串及用户名密码；建立数据库连接，得到 `Connection` 对象；获取数据库操作对象 `Statement`；执行 SQL 语句；处理数据库操作返回的结果；关闭对象，释放资源。

## 2. Hibernate

Hibernate 是基于 Java 的开源持久化中间件，它对 JDBC 做了轻量的封装，并且采用了 ORM 映射机制。Hibernate 负责实现 Java 对象和关系数据库之间的映射，把 SQL 语句传给数据库，并且把数据库返回的结果封装成对象。内部封装了 JDBC 访问数据库的操作，向上层应用提供了面向对象的数据库访问 API。Hibernate 以对象的形式操作数据，不用关心数据库种类（换数据库只要修改配置文件即可），提高了开发效率。下面通过实例来介绍 Hibernate 的使用。

### 【范例】

首先从官网下载 Hibernate，这里下载的是 Hibernate4.3.10 版本，将下载的文件 `hibernate-release-4.3.10.Final.zip` 解压到电脑中，其目录如图 5-1 所示。

名称	大小
..	
documentation	
lib	
project	
changelog.txt	357,904
hibernate_logo.gif	1,456
lgpl.txt	26,428

图 5-1 Hibernate 目录

`lib` 下的 `required` 文件里的 `jar` 包是开发中必需的，将其全部添加到项目，并配置为 `Library`。再将 `project` 下的 `etc` 文件夹中的 `hibernate.cfg.xml` 复制到源文件根目录下。此文件为 Hibernate 读取的核心配置文件，包含了数据库连接的相关信息，将其做如下修改：

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
```

```

<session-factory>
    <!--配置 MySQL 数据库的连接参数 -->
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect
</property>
    <!-- 驱动程序名 -->
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.
Driver</property>
    <!-- 数据库名称 -->
    <property name="hibernate.connection.url">jdbc:mysql://127.0.0.1:3306/
demo</property>
    <!-- 用户名 -->
    <property name="hibernate.connection.username">root</property>
    <!-- 密码 -->
    <property name="hibernate.connection.password">123456</property>
    <!-- 添加 DemoBean.hbm.xml 映射文件 -->
    <mapping resource="com/hjc/demo/hibernate/DemoBean.hbm.xml"/>
</session-factory>
</hibernate-configuration>

```

写一个 Javabean 来对应数据库中的表，创建 DemoBean，代码如下：

```

package com.hjc.demo;

import java.io.Serializable;
import java.util.Date;

public class DemoBean implements Serializable {
    /**
     *
     */
    private static final long serialVersionUID = -2753477901931164396L;

    public DemoBean() {

    }

    private long id;
    private String name;
    private String sex;
    private Date createtime;

```

```
public long getId() {
    return id;
}

public void setId(long id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getSex() {
    return sex;
}

public void setSex(String sex) {
    this.sex = sex;
}

public Date getCreatetime() {
    return createtime;
}

public void setCreatetime(Date createtime) {
    this.createtime = createtime;
}
}
```

创建文件 DemoBean.hbm.xml, Javabeen 对应的表的相关内容如下:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM "http://www.hibernate.org/dtd/hibernate-
mapping-3.0.dtd" >
<hibernate-mapping>
    <!-- 一个 class 标签对应一个实体类, name 属性指定实体类名称, table 属性指定关联的
    数据库表 -->
```



```

<class name="com.hjc.demo.DemoBean" table="t_demo1">
    <!-- 主键 -->
    <id name="id" column="id">
        <!-- 主键的生成策略 -->
        <generator class="native"></generator>
    </id>
    <!-- 其他属性, name 对应实体类的属性, column 对应关系型数据库表的列 -->
    <property name="name" column="name"></property>
    <property name="sex" column="sex"></property>
    <property name="createtime" column="createtime"></property>
</class>
</hibernate-mapping>

```

在 `hibernate.cfg.xml` 中引入这个 XML 文件, 让 Hibernate 加载时能找到这张表, 添加如下代码:

```

<mapping resource="com/hjc/demo/hibernate/DemoBean.hbm.xml"/>

```

然后就可以进行开发了, 创建 `HibernateDemo`, 代码如下:

```

package com.hjc.demo.hibernate;

import java.util.Collections;
import java.util.Date;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

import com.hjc.demo.DemoBean;

public class HibernateDemo {
    private SessionFactory sessionFactory;

    public static void main(String[] args) {
        HibernateDemo demo = new HibernateDemo();
    }
}

```



```

demo.run();
}

public void run() {
    // 构建 sessionFactory
    System.out
        .println("=====sessionFactory=====");
    sessionFactory = buildSessionFactory();
    // 添加/修改数据
    System.out
        .println("=====添加/修改数据=====");
    DemoBean bean = new DemoBean();
    bean.setId(1);
    bean.setName("何金成");
    bean.setSex("男");
    bean.setCreatetime(new Date());
    DemoBean bean2 = new DemoBean();
    bean2.setId(2);
    bean2.setName("何金成 2");
    bean2.setSex("男");
    bean2.setCreatetime(new Date());
    DemoBean bean3 = new DemoBean();
    bean3.setId(3);
    bean3.setName("何金成 3");
    bean3.setSex("男");
    bean3.setCreatetime(new Date());
    save(bean);
    save(bean2);
    save(bean3);
    // 查询数据 (一条)
    System.out
        .println("=====查询数据 (一条)=====");
    DemoBean findBean = find("where id=2");
    System.out.println(findBean.getId());
    System.out.println(findBean.getName());
    System.out.println(findBean.getSex());
    System.out.println(findBean.getCreatetime().toLocaleString());
    // 删除数据
    System.out.println("=====删除数据=====");
    DemoBean delBean = find("where id=3");

```

```

delete(delBean);
// 查询数据 (列表)
System.out
    .println("=====查询数据 (列表) =====");
List<DemoBean> list = list("where l=1");
for (DemoBean demoBean : list) {
    System.out.println("list=====");
    System.out.println(demoBean.getId());
    System.out.println(demoBean.getName());
    System.out.println(demoBean.getSex());
    System.out.println(demoBean.getCreatetime().toLocaleString());
}
System.out
    .println("=====释放资源=====");
sessionFactory.close();
}

/**
 * 查找数据 (一条)
 *
 * @param where
 * @return
 */
public DemoBean find(String where) {
    // 获取操作 session
    Session session = sessionFactory.openSession();
    // 开始事务
    Transaction tr = session.beginTransaction();
    DemoBean ret = null;
    try {
        String hql = "from DemoBean " + where;
        Query query = session.createQuery(hql);
        ret = (DemoBean) query.uniqueResult();
        // 提交事务
        tr.commit();
    } catch (Exception e) {
        // 回滚事务
        tr.rollback();
        e.printStackTrace();
    }
}

```

```
        return ret;
    }

    /**
     * 查询数据（列表）
     *
     * @param where
     * @return
     */
    public List<DemoBean> list(String where) {
        // 获取操作 session
        Session session = sessionFactory.openSession();
        // 开始事务
        Transaction tr = session.beginTransaction();
        List<DemoBean> list = Collections.EMPTY_LIST;
        try {
            String hql = "from DemoBean " + where;
            Query query = session.createQuery(hql);
            list = query.list();
            // 提交事务
            tr.commit();
        } catch (Exception e) {
            // 回滚事务
            tr.rollback();
            e.printStackTrace();
        }
        return list;
    }

    /**
     * 添加/修改数据
     *
     * @param bean
     */
    public void save(DemoBean bean) {
        // 获取操作 session
        Session session = sessionFactory.openSession();
        // 开始事务
        session.beginTransaction();
        try {
```



```
        session.saveOrUpdate(bean);
        // 提交事务
        session.getTransaction().commit();
        System.out.println("保存数据: ");
        System.out.println(bean.getId());
        System.out.println(bean.getName());
        System.out.println(bean.getSex());
        System.out.println(bean.getCreatetime().toLocaleString());
    } catch (Throwable e) {
        // 回滚事务
        session.getTransaction().rollback();
        e.printStackTrace();
    }
}

/**
 * 删除数据
 *
 * @param bean
 */
public void delete(DemoBean bean) {
    // 获取操作 session
    Session session = sessionFactory.openSession();
    // 开始事务
    session.beginTransaction();
    try {
        session.delete(bean);
        // 提交事务
        session.getTransaction().commit();
        System.out.println("删除数据: ");
        System.out.println(bean.getId());
        System.out.println(bean.getName());
        System.out.println(bean.getSex());
        System.out.println(bean.getCreatetime().toLocaleString());
    } catch (Throwable e) {
        // 回滚事务
        session.getTransaction().rollback();
        e.printStackTrace();
    }
}
```



```

    }

    /**
     * 构建 SessionFactory
     *
     * @return
     */
    public SessionFactory buildSessionFactory() {
        Configuration cfg = new Configuration().configure();
        StandardServiceRegistryBuilder ssrb = new StandardServiceRegistryBuilder()
            .applySettings(cfg.getProperties());
        ServiceRegistry service = ssrb.build();
        SessionFactory factory = cfg.buildSessionFactory(service);
        return factory;
    }
}

```

【运行结果】以上代码包含了 Hibernate 的 CRUD 的基本操作，代码中也做了相应的注释。

```

=====sessionFactory=====
四月 10, 2016 11:49:58 下午 org.hibernate.annotations.common.reflection.
java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {4.0.5.Final}
四月 10, 2016 11:49:58 下午 org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {4.3.10.Final}
四月 10, 2016 11:49:58 下午 org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
四月 10, 2016 11:49:58 下午 org.hibernate.cfg.Environment buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
四月 10, 2016 11:49:58 下午 org.hibernate.cfg.Configuration configure
INFO: HHH000043: Configuring from resource: /hibernate.cfg.xml
四月 10, 2016 11:49:58 下午 org.hibernate.cfg.Configuration
getConfigurationInputStream
INFO: HHH000040: Configuration resource: /hibernate.cfg.xml
四月 10, 2016 11:49:58 下午 org.hibernate.cfg.Configuration addResource
INFO: HHH000221: Reading mappings from resource: com/hjc/demo/hibernate/
DemoBean.hbm.xml
四月 10, 2016 11:49:58 下午 org.hibernate.cfg.Configuration doConfigure

```

```

INFO: HHH000041: Configured SessionFactory: null
四月 10, 2016 11:49:58 下午 org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl configure
WARN: HHH0000402: Using Hibernate built-in connection pool (not for
production use!)
四月 10, 2016 11:49:58 下午 org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl buildCreator
INFO: HHH0000401: using driver [com.mysql.jdbc.Driver] at URL [jdbc:mysql:
//127.0.0.1:3306/demo]
四月 10, 2016 11:49:58 下午 org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl buildCreator
INFO: HHH000046: Connection properties: {user=root, password=****}
四月 10, 2016 11:49:58 下午 org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl buildCreator
INFO: HHH000006: Autocommit mode: false
四月 10, 2016 11:49:58 下午 org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl configure
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
四月 10, 2016 11:49:59 下午 org.hibernate.dialect.Dialect <init>
INFO: HHH0000400: Using dialect: org.hibernate.dialect.MySQLDialect
四月 10, 2016 11:49:59 下午 org.hibernate.engine.transaction.internal.
TransactionFactoryInitiator initiateService
INFO: HHH0000399: Using default transaction strategy (direct JDBC
transactions)
四月 10, 2016 11:49:59 下午 org.hibernate.hql.internal.ast.
ASTQueryTranslatorFactory <init>
INFO: HHH0000397: Using ASTQueryTranslatorFactory
=====添加/修改数据=====
保存数据:
1
何金成
男
2016-4-10 23:49:59
保存数据:
2
何金成 2
男
2016-4-10 23:49:59
保存数据:
3

```

```

何金成 3
男
2016-4-10 23:49:59
=====查询数据（一条）=====
2
何金成 2
男
2016-4-10 23:49:59
=====删除数据=====
删除数据：
3
何金成 3
男
2016-4-10 23:49:59
=====查询数据（列表）=====
list=====
1
何金成
男
2016-4-10 23:49:59
list=====
2
何金成 2
男
2016-4-10 23:49:59
=====释放资源=====
四月 10, 2016 11:50:00 下午 org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl stop
INFO: HHH000030: Cleaning up connection pool [jdbc:mysql://127.0.0.1:
3306/demo]

```

**【代码解析】**Hibernate 作为一款成熟的 ORM 框架，使用起来非常简单。首先，需要配置好相应的配置文件，初始化时构建好 Hibernate 的 SessionFactory。然后，利用 SessionFactory 获取到操作的 Session，就可以通过 Session 的 API 来操作数据库了。如果做了事务控制，还需要控制好事务的回滚。

Hibernate 的操作步骤如图 5-2 所示。

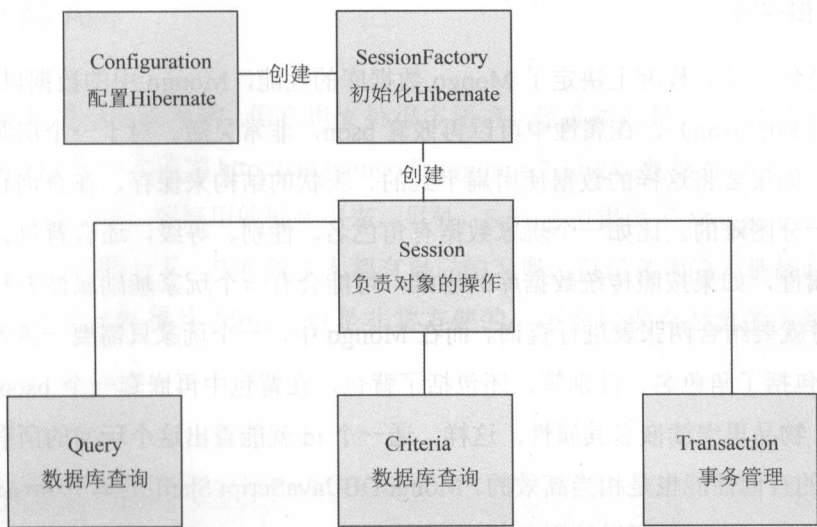


图 5-2 Hibernate 操作步骤

通过以上范例，就可以实现对 Hibernate 的简单操作，关于 Hibernate 的详细内容，请翻阅更多有关 Hibernate 的资料，这里不再赘述。个人认为，Hibernate 是关系型数据库开发中最得力的助手。

## 5.3 MongoDB 的介绍及使用

MongoDB 是文档型数据库，具有高性能、开源、文档型等特点。MongoDB 是当前 NoSQL 数据库中比较热门的一种。它在许多场景中可替代传统的关系型数据库或键/值存储方式。MongoDB 使用 C++ 开发。Mongo 的官方网址是 <http://www.mongodb.org/>，读者可以在此获得更详细的信息。

### 5.3.1 MongoDB 的主要特点

MongoDB 数据库作为众多 NoSQL 中最热门的一款数据库，在设计和实现上有很多特点，这些特点使 MongoDB 在很多项目中使用起来更得心应手，自然也就深受开发者的喜爱。



## 1. 文档存储

这一特性在某种程度上决定了 Mongo 数据库的性能, Mongo 中的数据以 bson 形式存储(二进制的 json), 在属性中可以再嵌套 bson, 非常灵活。对于一个层级式的数据结构来说, 如果要将这样的数据使用扁平式的、表状的结构来保存, 在查询和获取数据方面都是十分困难的。比如一个玩家数据有角色名、性别、等级, 还有背包、背包中物品、物品属性, 如果按照传统数据库的思维, 可能会有一个玩家基础属性表和一个背包表, 查询时就要结合两张表进行查询。而在 Mongo 中, 一个玩家只需要一条数据, 一个玩家 id 就包括了角色名、性别等, 还包括了背包, 在背包中再嵌套一个 bson 存储背包中的物品, 物品再继续嵌套其属性。这样, 通一个 id 就能查出这个玩家的所有数据, 而且 Mongo 的查询性能也是相当高效的。MongoDB JavaScript Shell 是基于 JavaScript 的解释器, 在 Mongo 中, 还可以使用 js 操作 Mongo。

## 2. 可扩展性

这是最适合游戏开发的特性之一, 游戏数据是多变的, 可能今天要在玩家数据结构中加一个属性, 明天又要在玩家数据结构中去掉一个属性。在关系型数据库中, 面对众多数据, 可能开发者就会发愁了, 但使用 Mongo 可以很容易地进行横向扩展。

## 3. 易查询

这和第一点有很大的联系, MongoDB 以文档的形式存储数据, 不支持事务和表连接。因此查询的编写、理解和优化都容易很多。简单查询设计思路不同于 SQL 模式, 嵌入文档在特定的环境下可进行更好的查询。游戏后端处理是要求效率的, 这一特点大大提升了 Mongo 在游戏后端的可用性。

## 4. 安全性

由于 MongoDB 客户端生成的查询为 BSON 对象, 而不是可以被解析的字符串, 所以可降低受到 SQL 注入的攻击的风险。不同于常见的 SQL 语句, 普通的 SQL 攻击自然对 bson 无效。

游戏中的数据需求经常发生变化, 需要进行横向或纵向扩展, 而传统的关系型数据, 往往满足不了这种变化多端的需求。上述四个特点使 MongoDB 非常适合游戏中数据持久化的需求。

### 5.3.2 了解 API

Mongo 是用 C++编写的,但它也支持很多语言,最重要的是,它也提供 Java API, Mongo Java API 官方文档为 <http://api.mongodb.org/java/2.11.2/>。既然有官方 API,那就一定有人封装了更方便、更好用的框架出来。另外, Spring 也提供了 Mongo 的封装,有兴趣的用户可以去了解一下,毕竟每个人都有自己的习惯,自己喜欢的才是最好的。使用官方提供的 Java API 操作 Mongo 也是非常方便的。下面以保存对象的方法为例进行介绍。

#### 【范例】

```
package com.hjc.demo.mongo;

import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;

import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBObject;
import com.mongodb.Mongo;
import com.mongodb.MongoClient;
import com.mongodb.MongoClientOptions;
import com.mongodb.MongoCredential;
import com.mongodb.ServerAddress;
import com.mongodb.util.JSON;

/**
 * 保存对象 API
 */
public class MongoAPIDemo {

    public static void main(String[] args) throws UnknownHostException {
        // 第一: 实例化 Mongo 对象, 连接 MongoDB 服务器, 包含所有的数据库
        // 默认构造方法, 默认是连接本机, 默认端口号是 27017
        // 获取 MongoCredential
        MongoCredential credentials = MongoCredential.createMongoCRCredential(
            "kidbear", "war", "123321".toCharArray());
```

```

List<MongoCredential> credentialsList = new ArrayList
<MongoCredential>();
credentialsList.add(credentials);
// 获取 DBOptions
MongoClientOptions.Builder build = new MongoClientOptions.Builder();
build.connectionsPerHost(50); // 与目标数据库能够建立的最大 connection
数量为 50
build.threadsAllowedToBlockForConnectionMultiplier(50); // 如果当前
所有的 connection 都在使用中, 则每个 connection 上可以有 50 个线程排队等待
build.maxWaitTime(120000);
build.connectTimeout(60000);
MongoClientOptions myOptions = build.build();
// 获取 ServerAddress
ServerAddress addr = new ServerAddress();
String hosts = "123.57.211.130:27017";
for (String host : hosts.split("&")) {
    String ip = host.split(":")[0];
    String port = host.split(":")[1];
    addr = new ServerAddress(ip, Integer.parseInt(port));
}
// new Mongo 实例
Mongo mongo = new MongoClient(addr, credentialsList, myOptions);
// 第二: 连接具体的数据库
// 其中, 参数是具体数据库的名称, 若服务器中不存在会自动创建
DB db = mongo.getDB("war");
// 第三: 操作具体的表
// 在 MongoDB 中没有表的概念, 而是指集合
// 其中, 参数是数据库中的表, 若不存在会自动创建
DBCollection collection = db.getCollection(MongoCollections.USER_DATA);
// 添加操作
// 在 MongoDB 中没有行的概念, 而是指文档
BasicDBObject document = new BasicDBObject();
document.put("id", 1);
document.put("name", "小明");
// 保存到集合中
collection.insert(document);
// 也可以保存这样的 json 串
/*
* { "id":1, "name","小明", "address": { "city":"beijing", "code":
"065000"

```

```

* } }
*/
// 实现上述 json 串的思路如下:
// 第一种: 类似 xml 时, 不断添加
BasicDBObject addressDocument = new BasicDBObject();
addressDocument.put("city", "beijing");
addressDocument.put("code", "065000");
document = new BasicDBObject();
document.put("address", addressDocument);
// 保存在数据库中
collection.insert(document);
// 第二种: 直接把 json 存到数据库中
String jsonTest = "{ 'id':1, 'name': '小明', "
    + " 'address': { 'city': 'beijing', 'code': '065000' } }";
DBObject dbobjct = (DBObject) JSON.parse(jsonTest);
collection.insert(dbojct);
}
}

```

【运行结果】执行完以上程序之后, 可以打开 MongoVUE, 看到我们插入了以下三条数据 (见图 5-3):

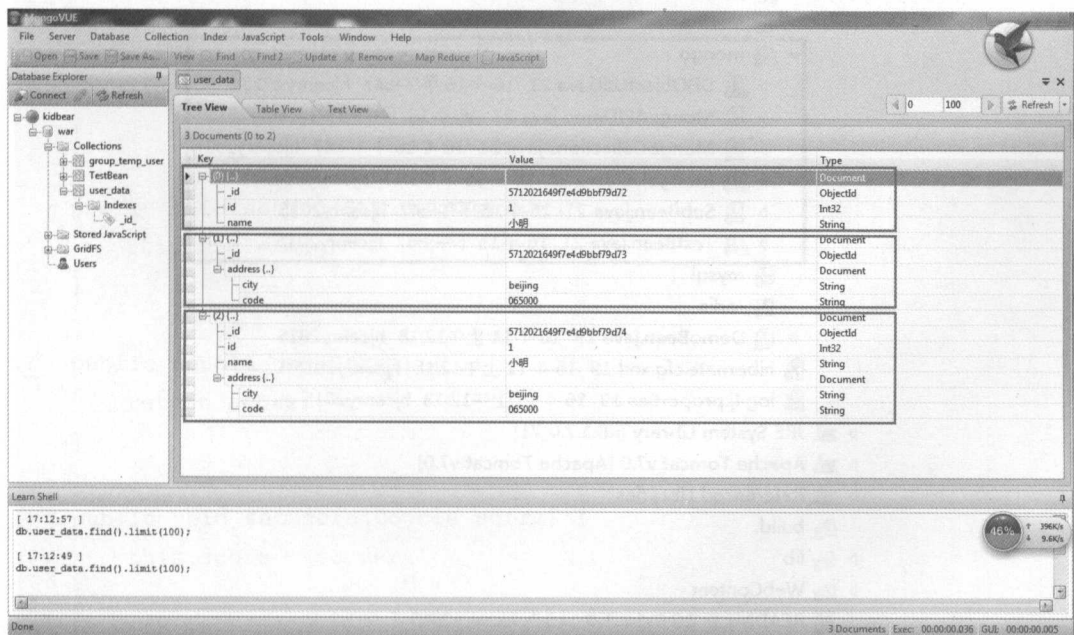


图 5-3 MongoVUE 查询



【代码解析】使用 Mongo 官方提供的 Java API，首先需要实例化 Mongo 对象，并设置好所有连接条件，然后通过 Mongo 的 `getDB` 连接具体的数据库 DB 实例，通过 DB 的 `getCollection` 方法可获得数据库中具体的集合 `DBCollection`，通过 `collection` 即可操作文档数据。

### 5.3.3 Mongo 的使用

为了使 Mongo 更符合实际项目中的使用情景，我对 Mongo 的操作做了一些简单的封装，使其调用起来更方便。当然，这些封装只针对 Mongo 的使用，还没有对缓存等做封装，这里主要是理解 Mongo 的使用。

#### 【范例】

首先添加官方的 jar 包到开发环境中，官方提供的 jar 包是 `mongo-java-drive.jar`，封装的工具类如图 5-4 所示。

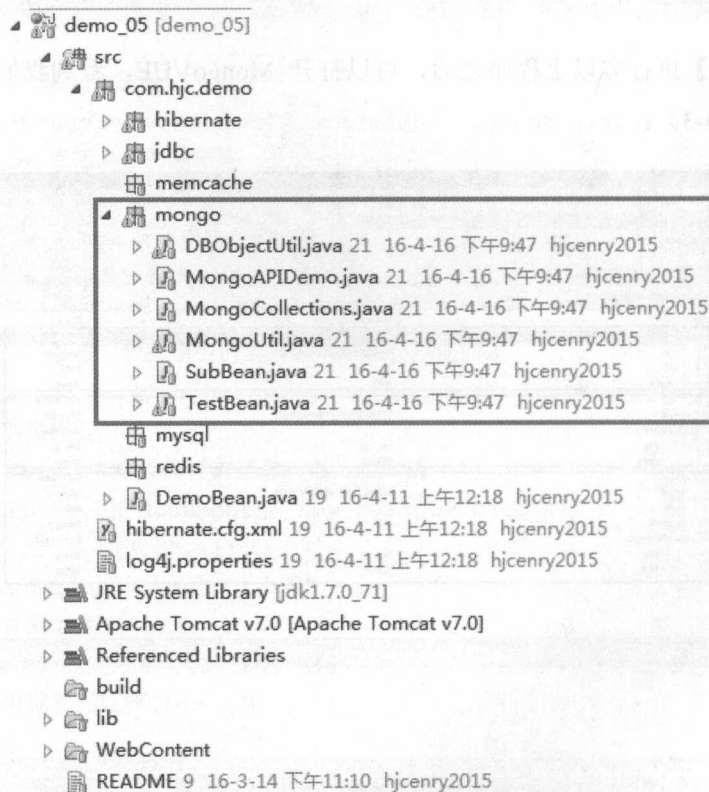


图 5-4 Mongo 的封装结构

其中, TestBean 作为要插入的数据 Javabean, SubBean 作为 TestBean 的一个子类, DBObjectUtil 为 DBObject 和 Javabean 的转换工具类, MongoAPIDemo 为 Mongo API 操作的演示示例, MongoCollections 为 Mongo 中表名的集合, MongoUtil 为数据库连接和操作的工具类。下面分别是这些类的源码。

## 1. TestBean

```
package com.hjc.demo.mongo;

public class TestBean {
    private Long id;
    private String msg;
    private Double score;
    private SubBean subBean;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public Double getScore() {
        return score;
    }

    public void setScore(Double score) {
        this.score = score;
    }

    public SubBean getSubBean() {
```

```
        return subBean;
    }

    public void setSubBean(SubBean subBean) {
        this.subBean = subBean;
    }
}
```

## 2. SubBean

```
package com.hjc.demo.mongo;

public class SubBean {
    private Long id;
    private String str;

    public String getStr() {
        return str;
    }

    public void setStr(String str) {
        this.str = str;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

## 3. DBObjectUtil

```
package com.hjc.demo.mongo;

import java.lang.reflect.InvocationTargetException;

import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
```

```

import com.mongodb.util.JSON;

/**
 * @author 何金成DBObject和JavaBean的相关转换,借助fastjson使json字符串作为
 * 两者转换的中间形式
 *
 */
public class DBObjectUtil {

    /**
     * 把实体bean对象转换成DBObject
     *
     * @param bean
     * @return
     * @throws IllegalArgumentException
     * @throws IllegalAccessException
     */
    public static <T> DBObject bean2DBObject(T bean) {
        if (bean == null) {
            return null;
        }
        DBObject dbObject = new BasicDBObject();
        // 使用fastjson转换javabean为json字符串
        String json = com.alibaba.fastjson.JSON.toJSONString(bean);
        // 使用Mongo提供的Json工具类转为DBObject对象
        dbObject = (DBObject) JSON.parse(json);
        return dbObject;
    }

    /**
     * 把DBObject转换成bean对象
     *
     * @param dbObject
     * @param bean
     * @return
     * @throws Exception
     * @throws InvocationTargetException
     * @throws NoSuchMethodException
     */
    @SuppressWarnings("unchecked")

```



```

    public static <T> T dbObject2Bean(DBObject dbObj, Class<T> clz)
        throws Exception {
        if (dbObj == null) {
            return null;
        }
        // 使用 fastjson 把 DBOBJECT 对象转化为 json 字符串
        String json = com.alibaba.fastjson.JSON.toJSONString(dbObj);
        // 使用 fastjson 把 json 字符串转换为 JavaBean
        T obj = com.alibaba.fastjson.JSON.parseObject(json, clz);
        return obj;
    }
}

```

#### 4. MongoCollections

```

package com.hjc.demo.mongo;

public class MongoCollections {
    public static final String USER_DATA = "user_data";// 用户数据表
}

```

#### 5. MongoUtil

```

package com.hjc.demo.mongo;

import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import com.mongodb.BasicDBList;
import com.mongodb.BasicDBObject;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.DBObject;
import com.mongodb.MapReduceOutput;
import com.mongodb.MongoClient;
import com.mongodb.MongoClientOptions;

```

```
import com.mongodb.MongoCredential;
import com.mongodb.ServerAddress;

/**
 * @ClassName: MongoUtil
 * @Description: mongo
 * @author 何金成
 * @date 2016年1月19日 下午3:35:25
 *
 */
public class MongoUtil {
    private MongoClient mongo = null;
    private DB db = null;
    private static final Map<String, MongoUtil> instances = new ConcurrentHashMap
<String, MongoUtil>();
    public static final String DB_ID = "id"; // DB 中 id 字段名
    public static final String DB = "war"; // DB 中 id 字段名

    /**
     * 实例化
     *
     * @return MongoDBManager 对象
     */
    static {
        getInstance("db"); // 初始化默认的 MongoDB 数据库
    }

    public static MongoUtil getInstance() {
        return getInstance("db"); // 配置文件默认数据库前缀为 db
    }

    public static MongoUtil getInstance(String dbName) {
        MongoUtil mongoMgr = instances.get(dbName);
        if (mongoMgr == null) {
            mongoMgr = buildInstance(dbName);
            if (mongoMgr == null) {
                return null;
            }
            instances.put(dbName, mongoMgr);
        }
    }
}
```

```
        return mongoMgr;
    }

    private static synchronized MongoUtil buildInstance(String dbName) {
        MongoUtil mongoMgr = new MongoUtil();
        try {
            mongoMgr.mongo = new MongoClient(getServerAddress(dbName),
                getMongoCredential(dbName), getDBOptions(dbName));
            mongoMgr.db = mongoMgr.mongo.getDB("war");
            System.out.println("connect to MongoDB success!");
            boolean flag = mongoMgr.db.authenticate("username",
                "123456".toCharArray());
            if (!flag) {
                System.out.println("MongoDB auth failed");
                return null;
            }
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
        return mongoMgr;
    }

    /**
     * 获取集合（表）
     *
     * @param collection
     */
    public DBCollection getCollection(String collection) {
        db.requestStart();
        DBCollection collect = db.getCollection(collection);
        return collect;
    }

    /**
     * 插入
     *
     * @param collection
     * @param o
     */
```



```

public void insert(String collection, DBObject o) {
    getCollection(collection).insert(o);
    db.requestDone();
}

/**
 * 删除
 *
 * @param collection
 * @param q
 * 查询条件
 */
public List<DBObject> delete(String collection, DBObject q) {
    getCollection(collection).remove(q);
    List<DBObject> list = find(collection, q);
    db.requestDone();
    return list;
}

/**
 * @Title: updateOne
 * @Description: 更新一条数据
 * @param collection
 * @param q
 * @param setFields
 * @return DBObject
 * @throws
 */
public DBObject updateOne(String collection, DBObject q, DBObject
setFields) {
    DBObject ret = getCollection(collection).findAndModify(q, setFields);
    db.requestDone();
    return ret;
}

/**
 * 查找集合所有对象
 *
 * @param collection
 */

```



```

public List<DBObject> findAll(String collection) {
    List<DBObject> list = getCollection(collection).find().toArray();
    db.requestDone();
    return list;
}

```

/\*

\* 由于 Mongo 的 API 太多, 可根据不同的条件增、删、改、查不同的数据, 并且支持 MapReduce  
 \* 操作, 这里就不把全部代码粘贴出来了, 可参考源码部分的 demo 代码, 以下就省略一些 API 操作  
 \*/

/\*\*MongoUtil 操作的示例\*\*/

```

public static void main(String[] args) {
    try {
        System.out
            .println("/////直接使用 DBObject 调用 MongoUtil /////");
        // API 使用
        System.out.println("=====insert=====");
        getInstance().insert(
            MongoCollections.USER_DATA,
            new BasicDBObject().append("name", "admin3")
                .append("type", "2").append("score", 70)
                .append("level", 2)
                .append("inputTime", System.currentTimeMillis()));
        System.out.println("=====update=====");
        getInstance().update(MongoCollections.USER_DATA,
            new BasicDBObject().append("status", 1),
            new BasicDBObject().append("status", 2));
        // == group start =====
        System.out.println("=====group=====");
        StringBuilder sb = new StringBuilder(100);
        sb.append("function(obj, out){out.count++;out.}.append("scoreSum")
            .append("+=obj.}.append("score").append(";out.}")
            .append("levelSum").append("+=obj.}.append("level")
            .append('}');
        String reduce = sb.toString();
        BasicDBList list = getInstance().group(
            MongoCollections.USER_DATA,
            new BasicDBObject("type", true),
            new BasicDBObject(),

```

```

        new BasicDBObject().append("count", 0)
            .append("scoreSum", 0).append("levelSum", 0)
            .append("levelAvg", (Double) 0.0), reduce,
        "function(out) { out.levelAvg = out.levelSum / out.count }");
for (Object o : list) {
    DBObject obj = (DBObject) o;
    System.out.println(obj);
}
// ===== group end=====
// === mapreduce start =====
System.out
    .println("=====mapreduce start=====");
List<DBObject> list2 = getInstance()
    .mapReduce(
        MongoCollections.USER_DATA,

        "function() {emit({type:this.type},{type:this.type,score:this.score,level:this.level});}",

        "function(key,values){var result={type:key.type,score:0,level:0};var count=0;values.forEach(function(value){result.score+=value.score;result.level+=value.level;count++});result.level=result.level/count;return result;}",

        "group_temp_user", new BasicDBObject(),
        new BasicDBObject("score", 1));
for (DBObject o : list2) {
    System.out.println(o);
}
// ===== mapreduce end=====
System.out.println("=====find all=====");
System.out.println(getInstance()
    .findAll(MongoCollections.USER_DATA));
System.out.println("=====find=====");
System.out.println(getInstance().find(
    MongoCollections.USER_DATA,
    new BasicDBObject("inputTime", new BasicDBObject("$gt",
        1348020002890L)),
    new BasicDBObject().append("_id", "-1"), 1, 2));
System.out.println("=====delete=====");
getInstance().delete(MongoCollections.USER_DATA,
    new BasicDBObject());

```

```

        System.out.println(getInstance().findAll(MongoCollections.
USER_DATA));
        System.out
            .println("/////JavaBean 与 DBObject 的转换封装使用/////");
        // 封装后的 MongoUtil 的使用
        // 声明一个 Bean
        TestBean bean1 = new TestBean();
        bean1.setId(11);
        bean1.setMsg("test bean1");
        bean1.setScore(200.01d);
        SubBean sub1 = new SubBean();
        sub1.setId(11);
        sub1.setStr("sub bean 1");
        List<SubBean> subBeans = new ArrayList<SubBean>();
        subBeans.add(sub1);
        bean1.setSubBean(sub1);
        // 利用 DBObjectUtil 工具类将 Javabean 转换成 DBObject
        System.out.println("=====insert=====");
        MongoUtil.getInstance().insert(MongoCollections.USER_DATA,
            DBObjectUtil.<TestBean>bean2DBObject(bean1));
        bean1.setId(21); // 修改 id 再插入一条数据
        MongoUtil.getInstance().insert(MongoCollections.USER_DATA,
            DBObjectUtil.bean2DBObject(bean1));
        BasicDBObject query = new BasicDBObject();
        query.put("id", 11); // 查询 id 为 1 的数据
        BasicDBObject setFields = new BasicDBObject();
        setFields.put("str", "test bean 1 update"); // 查询 id 为 1 的数据
        System.out.println("=====update=====");
        MongoUtil.getInstance().update(MongoCollections.USER_DATA, query,
            setFields);
        System.out.println("=====delete=====");
        BasicDBObject delObj = new BasicDBObject();
        delObj.put(DB_ID, 21);
        MongoUtil.getInstance().delete(MongoCollections.USER_DATA, delObj);
        System.out.println("=====find all=====");
        List<DBObject> findList = MongoUtil.getInstance().findAll(
            MongoCollections.USER_DATA);
        for (DBObject dbObject : findList) {
            TestBean findBean = new TestBean();
            System.out.println("for each list:");

```



```

        System.out.println(dbObject.toString());
        findBean = DBObjectUtil.<TestBean> dbObject2Bean(dbObject,
            TestBean.class);
        System.out.println(findBean.getId());
        System.out.println(findBean.getMsg());
        System.out.println(findBean.getScore());
        System.out.println(findBean.getSubBean().getId());
        System.out.println(findBean.getSubBean().getStr());
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

【运行结果】直接单击右键运行 MongoUtil，就可以看到 MongoUtil 中测试代码的输出，如下：

```

connect to MongoDB success!
////////////////直接使用 DBObject 调用 MongoUtil////////////////
=====insert=====
=====update=====
=====group=====
{ "type" : null , "count" : 1.0 , "scoreSum" : 200.01 , "levelSum" : NaN ,
"levelAvg" : NaN}
{ "type" : "2" , "count" : 1.0 , "scoreSum" : 70.0 , "levelSum" : 2.0 ,
"levelAvg" : 2.0}
=====mapreduce start=====
{ "value" : { "type" : "2" , "score" : 210.0 , "level" : 2.0}}
=====find all=====
[{"_id" : { "$oid" : "571224c749f74440e58ecfca"} , "id" : 1 , "msg" : "test
bean1" , "score" : 200.01 , "str" : "test bean 1 update" , "subBean" : { "id" :
2 , "str" : "sub bean 2"} , "subBeans" : [ { "$ref" : "$subBean" , "$id" :
null } , { } ] } , { "_id" : { "$oid" : "5712254d49f74dd10bb3b484"} , "name" :
"admin3" , "type" : "2" , "score" : 70 , "level" : 2 , "inputTime" :
1460806989054}]
=====find=====
[{"_id" : { "$oid" : "5712254d49f74dd10bb3b484"} , "name" : "admin3" ,
"type" : "2" , "score" : 70 , "level" : 2 , "inputTime" : 1460806989054}]
=====delete=====
[]

```



```

//////////JavaBean 与 DBObject 的转换封装使用//////////
=====insert=====
=====update=====
=====delete=====
=====find all=====
for each list:
{ "_id" : { "$oid" : "5712254d49f74dd10bb3b485"} , "id" : 1 , "msg" : "test
bean1" , "score" : 200.01 , "str" : "test bean 1 update" , "subBean" : { "id" :
1 , "str" : "sub bean 1"}}
1
test bean1
200.01
1
sub bean 1

```

**【代码解析】**通过对 Mongo 的 Java API 进行进一步的封装,使其使用起来更加规范,将创建 Mongo 连接、获取 Collection 等方法提出来作为公用方法,并提供插入、删除、查询等常用的接口,方便开发。

以上代码几乎包含了 Mongo 的所有操作示例。Mongo Java API 的核心思想就是使用建立连接的 Mongo 对象获取对应的 Collection,然后使用 DBObject 对象作为媒介,调用 Collection 的增、删、改、查操作,完成对 Mongo 数据库的操作。可能有人会问,使用官方提供的 Mongo Java API 是否也有封装了 Mongo Java API 的成熟框架?当然,这样的框架也是有很多的,个人觉得 Morphia 是最接近 Hibernate 的一款框架,它使 Mongo 的操作更加便捷。这里不做过多介绍,感兴趣的用户可以去网上搜索一下。不过因为 Mongo 数据是近期兴起的产品,所以我认为框架中还没有做得十分成熟的部分。但我相信,在目前这些框架中或在未来的某个框架中,一定会出现一款像 Hibernate 一样优秀的 Mongo 操作框架。

## 5.4 Memcache 的介绍及使用

Memcache 是内存数据库,也是一套分布式高速缓存系统,目前很多网站都使用 Memcache 来提升网站的访问速度,特别是大型网站及一些需要频繁访问数据库的网站。Memcache 能够十分明显地提升网站的访问速度,无论是在网站开发方面还是在游戏开

发方面，在应用服务器与数据库服务器之间添加一个缓存作为中间件是十分必要的，这既减少了数据库服务器的压力，又提升了数据访问的速度。Memcache 在内存空间里维护一张巨大的 hash 表，来存储各种格式的数据，包括图像、视频、文件、数据库检索结果等。

Memcache 会把数据调用到内存中，然后直接从内存读取数据，来提升读取的速度。当分配给 Memcache 的内存用完时，Memcache 将采用 LRU 算法（Least Recently Used，最近最少使用策略），先替换失效的数据，再替换最近没有使用的数据。通常 Memcache 会以 daemon（守护进程）的方式运行在服务器上，随时等待客户端的连接操作。

### 5.4.1 Memcache 的特点

Memcache 作为分布式缓存系统，使用非常广泛。Memcache 的优势基于其具有以下几个特点。

#### 1. Memory

Memcache 采用内存存储，因此读写速度快，但对内存的要求较高，所缓存的内容也不能持久化。Memcache 对 CPU 要求较低，通常采用 Memcache 服务端和一些 CPU 消耗高的、Memory 消耗低的应用部署在一个服务器。

#### 2. 集中式 Cache

Memcache 避开了分布式 Cache 传播的问题，但是如果一定需要保证可靠性，就需要 Cluster 的工作，多个 Memcache 可以作为一个虚拟的 Cluster，对 Cluster 的读写和对普通的 Memcache 的读写在性能上没有太大的差别。

#### 3. 分布式扩展

Memcache 采用可分布式扩展的模式，将部署在一台服务器上的多个 Memcache 的服务端或部署在多个服务器上的 Memcache 服务端组成虚拟的服务端，而这对于调用者是完全屏蔽和透明的，这样的分布式扩展提高了单机器的内存利用率。

#### 4. Socket 通信

虽然 Memcache 通常被放在内网作为 Cache 使用，但是传输内容大小和序列化的问

题仍然需要注意。Socket 的传输速率较高，当前支持 TCP 和 UDP 两种模式，同时可以根据客户端的不同，选择 NIO 同步或异步调用方式，但仍然需要注意序列化的成本和带宽的成本。如果对于同一类 Class 对象的序列化传输，第一次序列化时间较长，那么后续就会进行优化，序列化最大的消耗是类的序列化，而不是对象的序列化，如果只传输字符串，那是再好不过的，也省去了序列化的操作，因此在 Memcache 中保存的往往都是较小的内容。

## 5. 内存分配机制

Memcache 支持的最大存储对象大小为 1MB。虽然它的内存分配相对来说属于比较特殊的，但 Memcache 这种分配方式也是基于性能考虑的，简单的分配机制可以更容易地实现回收再分配，从而节省对 CPU 的使用。

## 6. Cache 机制

Memcache 的 Cache 机制很简单，它没有同步、消息分发或两阶段提交。Memcache 就是一个简单的 Cache，用户把内存存进去可以取出来。如果 Key 没有被命中，Memcache 会直接告诉用户这个 Key 没有对应任何内容，可以去数据库或其他地方取。而当用户在外部数据源取到内容时，就可以将内容直接放进 Cache，下次再使用这个 Key 时，就能取到内容了。同步数据有两种方式，修改之后立即更新 Cache 中的内容就会立即生效；或者容许一个失效时间，到了失效时间内容自动删除。后者通常用在一些实时性不高且写入不频繁的情况下。

### 5.4.2 Memcache 的使用场景

Memcache 的使用场景很多，通常有以下几种。

- (1) 小对象缓存，如用户 Token、权限、session、资源信息等。
- (2) 小的静态资源的缓存，如网站的首页一般可以使用 Memcache 进行缓存。
- (3) 数据库 SQL 结果集的缓存，对于数据库的负载有很大的缓解作用。

通常，应用服务器和数据库服务器的结构如图 5-5 所示。

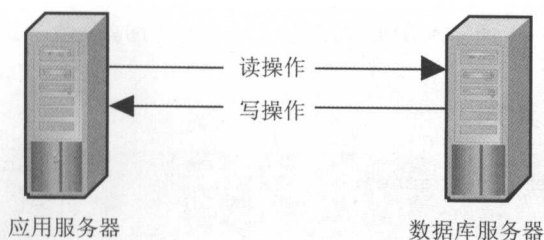


图 5-5 没有 Memcache 的结构图

这种结构下，无论是读操作还是写操作，一旦应用服务器的请求数增多，都将直接导致数据的访问量加大，甚至出现数据库服务器因无法负载而宕机的情况，不过在访问量不是很大的时候，这种情况一般不会出现。但在实际线上环境中，很难避免出现上述情况，而 Memcache 就能解决这种问题：用 Memcache 作为中间缓存，挡在数据库服务器的前面，分担数据库的读请求，实现读写分离。但是，当数据库中的数据发生变化时，要有一个很好的数据缓存同步策略，保持数据库与缓存的数据一致，如图 5-6 所示。

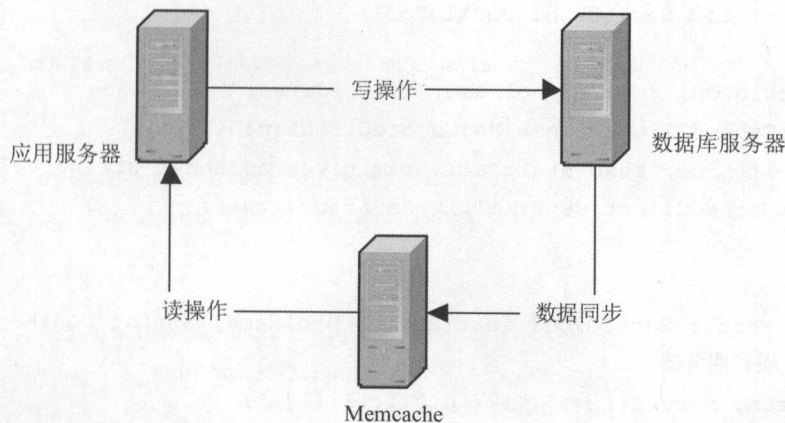


图 5-6 有 Memcache 的结构图

### 5.4.3 在 Java 中使用 Memcache

Memcache 的 Java 客户端有 `spymemcached`、`xmemcache` 和 `danga.Memcached`，这里以 `danga.Memcached` 为例介绍 Memcache 的操作。其中，`MemcachedCRUD` 是对 Memcache 的连接池和操作的管理类，`MemcacheClientDemo` 是操作示例代码，代码如下。



## 【范例】

Memcache 封装工具类:

```

package com.hjc.demo.memcache;

import java.util.Arrays;

import com.danga.MemCached.MemCachedClient;
import com.danga.MemCached.SockIOPool;

/**
 * @author 何金成
 */
public class MemcachedCRUD {
    public static String poolName = "gameDBPool";
    protected static MemCachedClient memCachedClient;
    protected static MemcachedCRUD memcachedCRUD = new MemcachedCRUD();
    public static SockIOPool sockIoPool;
    static {
        sockIoPool = init(poolName, "cacheServer");
        memCachedClient = new MemCachedClient(poolName);
        // if true, then store all primitives as their string value.
        memCachedClient.setPrimitiveAsString(true);
    }

    public static SockIOPool init(String poolName, String confKey) {
        // 缓存服务器
        String server[] = { "127.0.0.1:11211" };
        // 创建一个连接池
        SockIOPool pool = SockIOPool.getInstance(poolName);
        System.out.println("连接池 "+poolName+" 缓存配置 "+Arrays.toString(
(server)+"");
        pool.setServers(server); // 缓存服务器
        pool.setInitConn(50); // 初始化连接数
        pool.setMinConn(50); // 最小连接数
        pool.setMaxConn(500); // 最大连接数
        pool.setMaxIdle(1000 * 60 * 60); // 最大处理时间
        pool.setMaintSleep(3000); // 设置主线程睡眠时, 每 3 秒苏醒一次, 维持连接池大小
        pool.setNagle(false); // 关闭套接字缓存
        pool.setSocketTO(3000); // 连接建立后超时时间
    }
}

```

```
pool.setSocketConnectTO(0); // 连接建立时超时时间
pool.initialize();
return pool;
}

public static void destroy() {
    sockIoPool.shutdown();
}

MemcachedCRUD() {
}

public static MemcachedCRUD getInstance() {
    return memcachedCRUD;
}

private static final long INTERVAL = 100;

public boolean add(String key, Object o) {
    return memCachedClient.add(key, o);
}

public boolean update(String key, Object o) {
    return memCachedClient.replace(key, o);
}

public boolean saveObject(String key, Object msg) {
    boolean o = memCachedClient.keyExists(key);
    if (o) { // 如果 key 已经存在, 则用 msg 替换掉旧的 value
        return memCachedClient.replace(key, msg);
    } else {
        return memCachedClient.add(key, msg);
    }
}

public boolean keyExist(String key) {
    return memCachedClient.keyExists(key);
}

/**
```

```

    * delete
    *
    * @param key
    */
    public boolean deleteObject(String key) {
        return memCachedClient.delete(key);
    }

    public Object getObject(String key) {
        Object obj = memCachedClient.get(key);
        return obj;
    }

    public static MemCachedClient getMemCachedClient() {
        return memCachedClient;
    }
}

```

### Memcache 操作示例:

```

package com.hjc.demo.memcache;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class MemcacheClientDemo {

    public static void main(String[] args) {
        MemcacheClientDemo demo = new MemcacheClientDemo();
        demo.run();
    }

    public void run() {
        /** memcache add */
        System.out.println("=====memcache add=====");
        MemcachedCRUD memcache = MemcachedCRUD.getInstance();
        memcache.add("testKey1", 1);
        memcache.add("testKey2", "test value");
    }
}

```



```

List<String> list = new ArrayList<String>();
list.add("string1");
list.add("string2");
list.add("string3");
list.add("string4");
memcache.add("testKey3", list);
Map<Long, List<String>> map = new HashMap<Long, List<String>>();
map.put(11, list);
map.put(21, null);
map.put(31, list);
memcache.add("testKey4", map);
/** memcache get */
System.out.println("=====memcache get=====");
int val1 = Integer.parseInt((String) memcache.getObject("testKey1"));
System.out.println("val1:" + val1);
String val2 = (String) memcache.getObject("testKey2");
System.out.println("val2:" + val2);
List<String> val3 = (List<String>) memcache.getObject("testKey3");
System.out.print("val3:");
for (String string : val3) {
    System.out.print(string + ",");
}
System.out.println();
Map<Long, List<String>> val4 = (Map<Long, List<String>>) memcache
    .getObject("testKey4");
System.out.print("val4:");
for (Long key : val4.keySet()) {
    System.out.print("key:" + key + ",value:" + val4.get(key));
}
System.out.println();
/** memcache update */
System.out.println("=====memcache update=====");
memcache.saveObject("testKey2", "val after update");
String updateVal = (String) memcache.getObject("testKey2");
System.out.println("val2 after update:" + updateVal);
/** memcache key Exist */
System.out.println("=====memcache key Exist=====");
boolean flag = memcache.keyExist("testKey2");
System.out.println("is testKey2 exist:" + flag);
/** memcache delete */

```



```

        System.out.println("=====memcache delete=====");
        memcache.deleteObject("testKey3");
        boolean delFlag = memcache.keyExist("testKey3");
        System.out.println("is testKey3 exist after delete:" + delFlag);
    }
}

```

### 【运行结果】

```

连接池 gameDBPool 缓存配置[123.57.211.130:11211]
=====memcache add=====
=====memcache get=====
val1:1
val2:val after update
val3:string1,string2,string3,string4,
val4:key:1,value:[string1, string2, string3, string4]key:2,value:nullkey:3,
value:[string1, string2, string3, string4]
=====memcache update=====
val2 after update:val after update
=====memcache key Exist=====
is testKey2 exist:true
=====memcache delete=====
is testKey3 exist after delete:false

```

**【代码解析】**通过 SockIOPool 创建连接池连接 Memcache，通过 MemcacheClient 调用 API 进行具体的 Memcache 操作，MemcachedCRUD 是对 MemcacheClient 的更上一层的封装，使操作更加简便。

## 5.4.4 客户端使用要点

Memcache 具有高效、稳定、高内存利用率等特点，而且作为内存数据库，还具有读写速度快的特点。虽然如此，它也不是万能的。既然使用在内存中，那么数据就一定是不可靠的，根据场景使用 Memcache，才能发挥它最大的功效。

因此，客户端设计得是否合理十分重要，同时也给使用者提供了很大的空间去扩展和设计客户端来满足各种场景的需要，如容错、权重、效率、特殊的功能性需求、嵌入框架等。

## 5.5

## Redis 的介绍及使用

Redis 也是开源的内存数据库，与 Memcache 有很多相似之处，也有许多不同之处，Redis 有比 Memcache 更先进的 Key-value 存储系统。Redis 是一套构建高性能与可扩展的应用服务器程序的完美解决方案。

### 5.5.1 Redis 的特点

Redis 与其他内存数据库有很多相似性，也有自身独特的特点。Redis 自身的特点和优势使它受到开发者的青睐。Redis 主要有以下几个特点。

#### 1. 速度快

与 Memcache 一样，Redis 也是运行在内存中的数据库，因此它的执行速度也异常快，每秒能执行约 11 万的集合，每秒约 81000+条记录。

#### 2. 数据类型丰富

Redis 的存储系统由 Key-value 映射的字典组成，与其他非关系型数据库的不同在于：Redis 中值的类型不仅限于字符串，还支持以下五种值的类型。

(1) string: 最基础的数据类型，是二进制安全的，可存储任意内容，上限为 512MB。

(2) list: 链表结构，提供 pop、push、获取范围内的值等功能。

(3) set: 集合，与数学中的集合概念相同，无序目、无重复元素。

(4) sorted set: 有序集合，每个元素关联一个 score，以提供排序依据。

(5) hash: 字符串与字符串之间的映射。

合理运用 Redis 的数据类型，可以让 Redis 的使用充满无限可能。

#### 3. 操作原子性

所有的 Redis 操作都是原子的，保证多个客户端并发访问时获取到 Redis 服务器的值为最新值。

#### 4. 持久化

与 Memcache 不同的是, Redis 提供了数据持久化的功能, 提供 AOF 和 RDB 两种持久化模式。

#### 5. 应用场景丰富

Redis 有非常多的应用场景, 例如: 缓存、消息、队列 (Redis 原生支持订阅/发布)、数据库等。

### 5.5.2 Redis 的持久化

Redis 提供 AOF 和 RDB 两种持久化模式。

#### 1. AOF (Append Only File, 只追加数据到文件)

AOF 模式指 Redis 在执行过程中会把所有的写指令记录下来, 当数据恢复时, 再按照当时记录的顺序执行一遍命令。在 `redis.conf` 中添加 `appendonly yes` 就能打开 AOF 功能, 任何写操作进来时, Redis 都会追加到 AOF 文件末尾。

#### 2. RDB (Redis Database)

RDB 模式是在不同的时间点将 Redis 存储的数据快照存储到磁盘等介质上, Redis 在持久化过程中, 会先将数据写到临时文件中, 持久化过程结束之后再用临时文件替换上次持久化好的文件。Redis 会单独 fork 一个子进程进行持久化, 而主进程是不会进行任何 IO 操作的, 这也确保了在持久化的同时保留了 Redis 的性能。

### 5.5.3 Redis 的主从复制

Redis 提供了主从同步, 也支持一主多从及多级从结构。主从结构不仅可以备份数据到从服务器, 更可以承担一部分主服务器的读操作, 提高了 Redis 的读性能。Redis 的主从同步是异步执行的, 因此主从同步并不会减少 Redis 的运行性能。给 Redis 主服务器添加一个从服务器, 只需要在从服务器的 `redis.conf` 中添加 `slaveof masterip masterport` 命令, 先启动主服务器, 再启动从服务器, 从服务器会根据配置中的主服务器的 IP 端口对主服务器发出同步命令, 然后进行数据同步。

多组主从同步及主从切换可以通过 Redis Sentinel 进行管理, Sentinel 能对 Redis 集群中的主服务器及从服务器进行管理, 当主节点宕机时, Sentinel 能自动切换到从服务器。

## 5.5.4 在 Java 中使用 Redis

Java 提供了 Jedis 客户端对 Redis 进行操作, Demo 中封装了一个 Redis.java, 提供 Jedis 的各种操作 API, 可参照使用。RedisClientDemo 提供了 Redis 的使用测试代码, 代码如下。

### 【案例】

Redis 封装工具类:

```
package com.hjc.demo.redis;

import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.lang.reflect.InvocationTargetException;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.apache.commons.beanutils.BeanUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.Tuple;

public class Redis {
    private static Redis instance;
    public static Logger log = LoggerFactory.getLogger(Redis.class);
    public static final int DB = 0; // 默认数据库
```



```
public static String password = null;

public static Redis getInstance() {
    if (instance == null) {
        instance = new Redis();
        instance.init();
    }
    return instance;
}

private JedisPool pool;
public String host;
public int port;

/**
 * 测试用，不外调
 *
 * @return
 */
public Jedis getRedis() {
    return this.pool.getResource();
}

private void init() {
    String redisServer = "123.57.211.130:6379";
    redisServer = redisServer.trim();
    String[] tmp = redisServer.split(":");
    host = tmp[0];
    port = Integer.parseInt(tmp[1]);
    if (tmp.length == 2) {
        port = Integer.parseInt(tmp[1].trim());
    }
    log.info("Redis at {}:{}", host, port);
    // log.info("Redis sentinel at {}:{}", host, port);
    /** Redis 的使用 */
    pool = new JedisPool(host, port);
    /** Redis Sentinel 进行集群管理的使用 */
    // Set sentinels = new HashSet();
    // sentinels.add(new HostAndPort(host, port).toString());
    // pool = new JedisPool("master1", sentinels);
}
```

```
}

public void select(int index) {
    Jedis j = pool.getResource();
    // j.auth(password);
    j.select(index);
    pool.returnResource(j);
}

public Long hdel(int db, String key, String... fields) {
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    Long cnt = redis.hdel(key, fields);
    this.pool.returnResource(redis);
    return cnt;
}

public Map<String, String> hgetAll(int db, String key) {
    if (key == null) {
        return null;
    }
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    Map<String, String> ret = redis.hgetAll(key);
    this.pool.returnResource(redis);
    return ret;
}

public String hget(int db, String key, String field) {
    if (key == null) {
        return null;
    }
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    String ret = redis.hget(key, field);
    this.pool.returnResource(redis);
    return ret;
}
```

```
}

public void hset(int db, String key, String field, String value) {
    if (field == null || field.length() == 0) {
        return;
    }
    if (value == null || value.length() == 0) {
        return;
    }
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    redis.hset(key, field, value);
    this.pool.returnResource(redis);
}

public void set(int db, String key, String value) {
    if (value == null || key == null) {
        return;
    }
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    redis.set(key, value);
    this.pool.returnResource(redis);
}

public String get(int db, String key) {
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    String ret = redis.get(key);
    this.pool.returnResource(redis);
    return ret;
}

/**
 * 添加元素到集合中
 *
 * @param key
```



```

    * @param element
    */
    public boolean sadd(int db, String key, String... element) {
        if (element == null || element.length == 0) {
            return false;
        }

        Jedis redis = this.pool.getResource();
        // redis.auth(password);
        redis.select(db);
        boolean success = redis.sadd(key, element) == 1;
        this.pool.returnResource(redis);
        return success;
    }

    /**
     * 删除指定 set 内的元素
     */
    public boolean sremove(int db, String key, String... element) {
        if (element == null) {
            return false;
        }

        Jedis redis = this.pool.getResource();
        // redis.auth(password);
        redis.select(db);
        boolean success = (redis.srem(key, element) == 1);
        this.pool.returnResource(redis);
        return success;
    }

    public Set<String> sget(int db, String key) {
        Jedis redis = this.pool.getResource();
        // redis.auth(password);
        redis.select(db);
        Set<String> m = redis.smembers(key);
        this.pool.returnResource(redis);
        return m;
    }

    /**

```



```
* add by wangzhuan
*
* @Title: lrange
* @Description:
* @param key
* @param start
* @param end
* @return
*/
public List<String> lrange(int db, String key, int start, int end) {
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    List<String> list = redis.lrange(key, start, end);
    this.pool.returnResource(redis);
    return list;
}

public List<String> lgetList(int db, String key) {
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    long len = redis.llen(key);
    List<String> ret = redis.lrange(key, 0, len - 1);
    this.pool.returnResource(redis);
    return ret;
}

/**
 * 列表 list 中是否包含 value
 *
 * @param key
 * @param value
 * @return
 */
public boolean lexist(int db, String key, String value) {
    List<String> list = lgetList(db, key);
    return list.contains(value);
}
```

```

public List<String> lgetList(int db, String key, long len) {
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    long max = redis.llen(key);
    long l = max > len ? len : max;
    List<String> ret = redis.lrange(key, 0, l - 1);
    this.pool.returnResource(redis);
    return ret;
}

public Long del(int db, String key) {
    Jedis redis = this.pool.getResource();
    // redis.auth(password);
    redis.select(db);
    Long cnt = redis.del(key);
    this.pool.returnResource(redis);
    return cnt;
}

/**
 * 判断某一个 key 值的存储结构是否存在
 *
 * @Title: exist_
 * @Description:
 * @param key
 * @return
 */
public boolean exist_(int db, String key) {
    Jedis redis = this.pool.getResource();
    // //// redis.auth(password);
    redis.select(db);
    boolean yes = redis.exists(key);
    this.pool.returnResource(redis);
    return yes;
}

public long zadd(int db, String key, int score, String member) {
    Jedis redis = this.pool.getResource();
    // //// redis.auth(password);

```

```

        redis.select(db);
        long ret = 0;
        try {
            ret = redis.zadd(key, score, member);
        } catch (Exception e) {
            log.error(e.getMessage(), e);
        } finally {
            this.pool.returnResource(redis);
        }
        return ret;
    }

    /**
     * 按照 Redis 中有序集合的 score 的值从小到大排序, 返回 member 的排名, 排序从 0 开始
     *
     * @Title: zrank
     * @Description:
     * @param key
     * @param member
     * @return 设置名次从 1 开始。返回值为-1, 表示 member 无记录
     */
    public long zrank(int db, String key, String member) {
        Jedis redis = this.pool.getResource();
        // redis.auth(password);
        redis.select(db);
        long ret = -1;
        Long vv = redis.zrank(key, member);
        if (vv != null) {
            ret = vv.longValue();
        }
        this.pool.returnResource(redis);
        if (ret != -1) {
            ret += 1;
        }
        return ret;
    }

    public Set<String> zrange(int db, String key, long min, long max) {
        Jedis redis = this.pool.getResource();
        // redis.auth(password);

```



```

        redis.select(db);
        Set<String> ss = redis.zrange(key, min, max);
        this.pool.returnResource(redis);
        return ss;
    }

    /**
     * 删除 key 中的 member
     *
     * @Title: zrem
     * @Description:
     * @param key
     * @param mem
     * @return
     */
    public long zrem(int db, String key, String member) {
        Jedis redis = this.pool.getResource();
        // redis.auth(password);
        if (redis == null) {
            return -1;
        }
        redis.select(db);
        long result = -1;
        try {
            result = redis.zrem(key, member);
        } catch (Exception e) {
            log.error(e.getMessage(), e);
        } finally {
            this.pool.returnResource(redis);
        }
        return result;
    }

    public static void destroy() {
        getInstance().pool.destroy();
    }

    public void lrem(int db, String key, int count, String value) {
        Jedis redis = this.pool.getResource();
        // redis.auth(password);

```



```

        redis.select(db);
        redis.lrem(key, count, value);
        this.pool.returnResource(redis);
    }

    /**
     * 用于 Redis 的 API 太多, 这里不全部列出, 关于五种数据类型和更多的操作 API, 这里不
     再一一列出
     */
}

```

### Redis 客户端操作示例:

```

package com.hjc.demo.redis;

import java.util.List;
import java.util.Map;
import java.util.Set;

public class RedisDemo {

    public static void main(String[] args) {
        RedisDemo demo = new RedisDemo();
        demo.run();
    }

    public void run() {
        Redis redis = Redis.getInstance();
        /** redis save **/
        System.out.println("=====redis save=====");
        // string save
        System.out.println("string save:调用 set 时, 若 key 不存在则添加 key, 否
则修改 key 对应的值");
        redis.set(Redis.DB, "testKey1", "test string val1");
        // set save
        System.out.println("set save:set 中的元素不允许出现重复且无序");
        redis.sadd(Redis.DB, "testKey2", "test set val1");
        redis.sadd(Redis.DB, "testKey2", "test set val2");
        redis.sadd(Redis.DB, "testKey2", "test set val3");
        // hash save
        System.out

```

```

        .println("hash save:调用 hset 时,若 key 不存在则创建 key,若 hash
中不存在这个 hashkey 则修改其值,不存在则添加一条 hash 数据");
        redis.hset(Redis.DB, "testKey3", "hashKey1", "hashVal1");
        redis.hset(Redis.DB, "testKey3", "hashKey2", "hashVal2");
        redis.hset(Redis.DB, "testKey3", "hashKey3", "hashVal3");
        redis.hset(Redis.DB, "testKey3", "hashKey4", "hashVal4");
        // list save
        System.out.println("list save:数据在链表中是有序的,并可以重复添加数据");
        redis.lpush_(Redis.DB, "testKey4", "test list val1");
        redis.lpush_(Redis.DB, "testKey4", "test list val2");
        redis.lpush_(Redis.DB, "testKey4", "test list val3");
        // sorted set save
        System.out.println("sorted set save:有序 set 中的元素是有序的");
        redis.zadd(Redis.DB, "testKey5", 1, "test zset val1");
        redis.zadd(Redis.DB, "testKey5", 2, "test zset val2");
        redis.zadd(Redis.DB, "testKey5", 3, "test zset val3");
        redis.zadd(Redis.DB, "testKey5", 4, "test zset val4");
        /** redis get */
        System.out.println("=====redis get=====");
        // string get
        String stringRet = redis.get(Redis.DB, "testKey1");
        System.out.println("string get:" + stringRet);
        // set get
        Set<String> setRet = redis.sget(Redis.DB, "testKey2");
        System.out.print("set get:");
        for (String string : setRet) {
            System.out.print(string + ",");
        }
        System.out.println();
        // hash get
        String hashKeyRet = redis.hget(Redis.DB, "testKey3", "hashKey2");
        System.out.println("hash key get:" + hashKeyRet);
        Map<String, String> hashRet = redis.hgetAll(Redis.DB, "testKey3");
        System.out.print("hash get:");
        for (String string : hashRet.keySet()) {
            System.out.print("key[" + string + "]" + "value["
                + hashRet.get(string) + "],");
        }
        System.out.println();
        // list get

```

```

List<String> listRet = redis.lgetList(Redis.DB, "testKey4");
System.out.print("list get:");
for (String string : listRet) {
    System.out.println(string + ",");
}
// zset get
long val2Rank = redis.zrank(Redis.DB, "testKey5", "test zset val2");
System.out.println("zset get val2 rank:" + val2Rank);
Set<String> zsetRet = redis.zrange(Redis.DB, "testKey5", 0, 3);
System.out.print("zset get range:");
for (String string : zsetRet) {
    System.out.println(string + ",");
}
/** redis delete */
System.out.println("=====redis delete=====");
// string delete
System.out
    .println("string delete:调用 Redis 的 del 方法, 可直接删除 key,
所有的数据类型都可以通过这种方式直接删除整个 key");
redis.del(Redis.DB, "testKey1");
// set delete
System.out.println("set delete:删除 set 中的 val3");
redis.sremove(Redis.DB, "testKey2", "test set val3");
// hash delete
System.out.println("hash delete:删除 hash 中 key 为 hashKey4 的元素");
redis.hdel(Redis.DB, "testKey3", "hashKey4");
// list delete
System.out
    .println("list delete:删除 list 中值为 test list val3 的元素,
其中, count 参数为 0 代表删除全部, 正数代表正向删除 count 个此元素, 负数代表负向删除
count 个此元素");
redis.lrem(Redis.DB, "testKey4", 0, "test list val3");
// zset delete
System.out.println("zset delete:同 set 删除元素的方式相同");
redis.zrem(Redis.DB, "testKey5", "test zset val4");
System.out
    .println("除了以上常用 API 之外, 还有许多 API, 在 Redis 类中都有列出,
请参考 Redis 类, 或直接参照 Jedis 的官方文档");
}
}

```



**【运行结果】**以上 Demo 代码经过查询、修改和删除之后，控制台输出如下：

```

=====redis save=====
string save:调用 set 时，若 key 不存在则添加 key，否则修改 key 对应的值
set save:set 中的元素不允许出现重复且无序
hash save:调用 hset 时，若 key 不存在则创建 key，若 hash 中存在这个 hashkey 则修改其值，不存在则添加一条 hash 数据
list save:数据在链表中是有序的，并可以重复添加数据
sorted set save:有序 set 中的元素是有序的

=====redis get=====
string get:test string val1
set get:test set val3,test set val2,test set val1,
hash key get:hashVal2
hash
get:key[hashKey1]value[hashVal1],key[hashKey2]value[hashVal2],key[hashKey3]value[hashVal3],key[hashKey4]value[hashVal4],
list get:test list val3,
test list val2,
test list val1,
zset get val2 rank:2
zset get range:test zset val1,
test zset val2,
test zset val3,
test zset val4,

=====redis delete=====
string delete:调用 Redis 的 del 方法，可直接删除 key，所有的数据类型都可以通过这种方式直接删除整个 key
set delete:删除 set 中的 val3
hash delete:删除 hash 中 key 为 hashKey4 的元素
list delete:删除 list 中值为 test list val3 的元素，其中，count 参数为 0 代表删除全部，正数代表正向删除 count 个此元素，负数代表负向删除 count 个此元素
zset delete:同 set 删除元素的方式相同
除了以上常用 API 之外，还有许多 API，在 Redis 类中都有列出，请参考 Redis 类，或直接参照 Jedis 的官方文档

```

RedisDesktopManager 中的数据如图 5-7~图 5-11 所示。



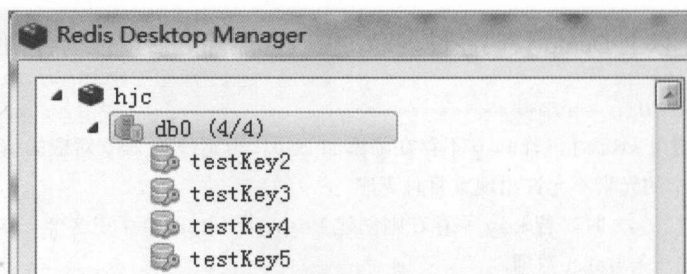


图 5-7 redis 查询 (testKey1 已被删除)

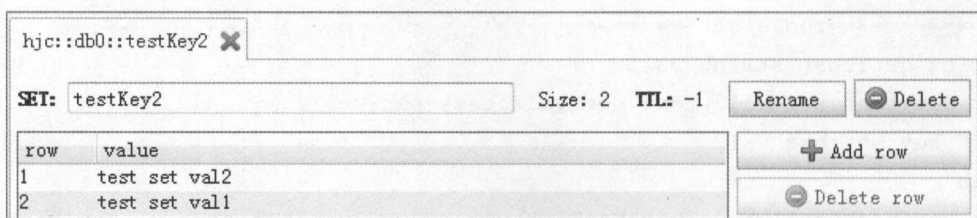


图 5-8 redis set 查询 (test set val3 已被删除)

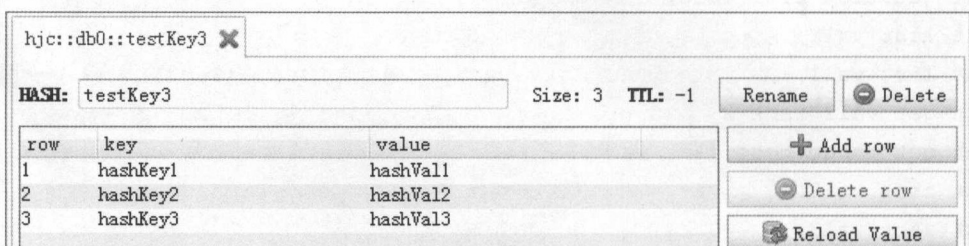


图 5-9 redis hash 查询 (hashVal4 已被删除)

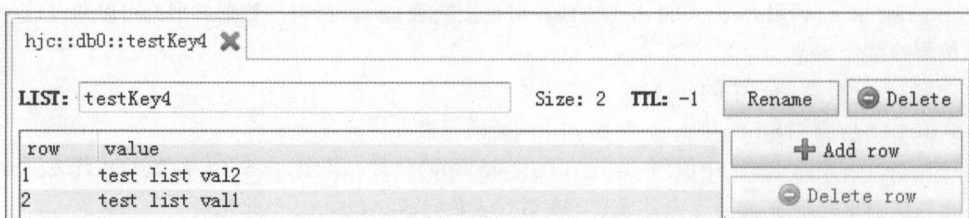


图 5-10 redis list 查询 (test list val3 已被删除)

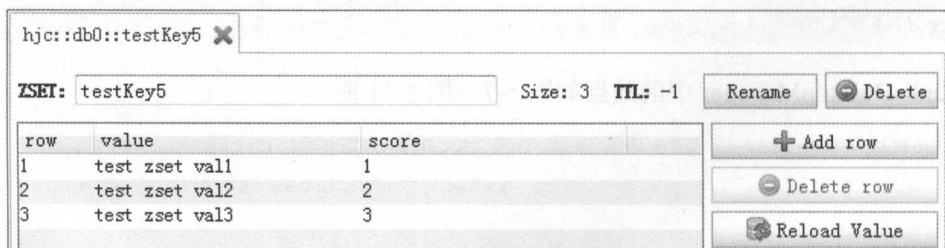


图 5-11 redis zset 查询 (test zset val4 已被删除)

**【代码解析】**使用 Jedis 连接 Redis，如果是单点 Redis，可以直接实例化一个设置好参数的 JedisPool 对象，并通过 JedisPool 获取 Jedis 对象来调用相应的 API；如果是连接 Sentinel 集群管理，可以通过实例化 SentinelPool 来获取 Jedis 对象调用相应的 API，Redis 类对 Redis 的操作做了更好的封装，使用起来更方便。

## 5.6 总结

在游戏服务器开发中，游戏数据的缓存或持久化，都需要一个高可用的方案，以保证服务器中数据的稳定。不管选择的是 Memcached 还是 Redis，是 MySQL 还是 Mongo，只要应用场景合理，就是一个可行的方案。合理使用各种数据，才能让服务器的游戏数据更加稳定可靠。

# 第6章

## 游戏逻辑

### 6.1

### 逻辑架构

游戏中最重要的部分就是逻辑处理部分，这也是每个游戏最独特的部分。不同的游戏，可能会有不同的逻辑架构。从用户请求接入到用户消息返回，是一个完整的逻辑流程。各游戏的逻辑架构都不大一样，本章以我常用的一种架构为例进行讲解。

逻辑流程指的是从用户请求接入，到请求处理，最后返回请求结果的一整套流程。

#### 6.1.1 项目目录

在设计逻辑框架之前要创建项目，并确定项目大致的目录结构，如图 6-1 所示。

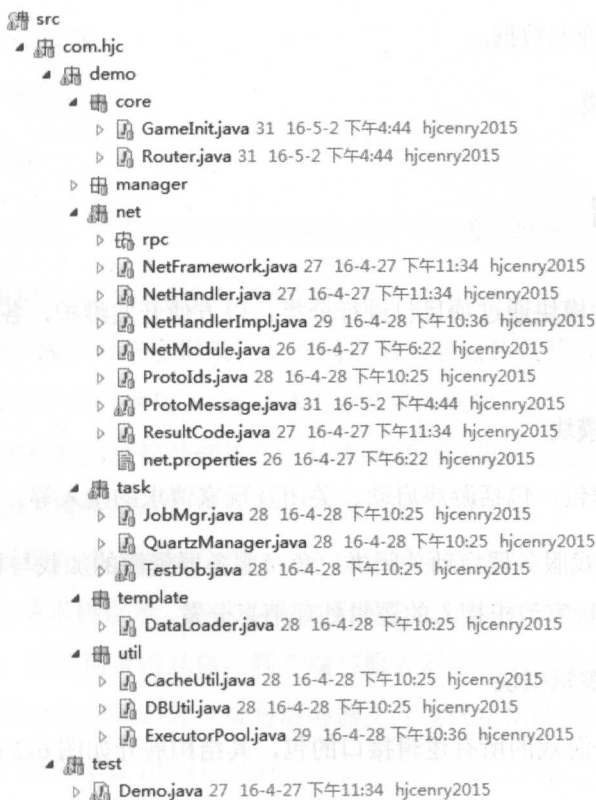


图 6-1 项目结构

项目通过 maven 进行搭建，选择 maven-archetype-webapp 作为原型进行开发，选择 spring 作为整合框架，并且通过 tomcat 进行打包发布。游戏服务器的通信其实可以不使用 tomcat 作为容器，我们在游戏中会通过 Netty 来实现网络通信架构，这里使用 Web 框架，主要是方便开发中打包、编译及发布。

maven 框架搭建好之后，会有 src/main/java、src/main/resources 等文件夹，java 代码如游戏逻辑等放在 src/main/java 中，配置文件如数据库配置等放在 src/main/resources 中。在 src/main/java 目录下，建立的项目目录如图 6-1 所示，主要有以下几个包。

(1) core: 核心部分。

(2) manager: 逻辑模块。

(3) net: 网络通信部分。

(4) task: 定时任务。



(5) template: 静态数据。

(6) util: 工具类。

## 6.1.2 模块介绍

游戏中的各个大模块通过项目包进行分类, 以方便开发维护, 各个模块的功能在其目录中继续细分。

### 1. core: 核心模块

游戏的核心处理包, 包括游戏启动、关闭及玩家请求的接入等。

➤ GameInit: 游戏服务器启动关闭类, 负责服务器资源的加载与释放。

➤ Router: 处理玩家请求接入的逻辑处理消息分发。

### 2. manager: 逻辑模块

逻辑模块是整个游戏的所有逻辑接口的包, 其结构展开如图 6-2 所示。

```

manager
├── event
│   ├── ED.java 31 16-5-2 下午4:44 hjcenry2015
│   ├── Event.java 31 16-5-2 下午4:44 hjcenry2015
│   ├── EventMgr.java 31 16-5-2 下午4:44 hjcenry2015
│   └── EventProc.java 30 16-4-28 下午10:42 hjcenry2015
├── module1
│   └── Demo1Mgr.java 31 16-5-2 下午4:44 hjcenry2015
├── module2
│   └── Demo2Mgr.java 31 16-5-2 下午4:44 hjcenry2015
├── module3
│   └── Demo3Mgr.java 31 16-5-2 下午4:44 hjcenry2015
├── module4
│   └── Demo4Mgr.java 31 16-5-2 下午4:44 hjcenry2015
├── module5
│   └── Demo5Mgr.java 31 16-5-2 下午4:44 hjcenry2015
├── module6
│   └── Demo6Mgr.java 31 16-5-2 下午4:44 hjcenry2015
├── module7
│   └── Demo7Mgr.java 31 16-5-2 下午4:44 hjcenry2015
├── module8
│   ├── Demo8Mgr.java 31 16-5-2 下午4:44 hjcenry2015
│   └── DemoBean.java 31 16-5-2 下午4:44 hjcenry2015
└── module9
    └── Demo9Mgr.java 31 16-5-2 下午4:44 hjcenry2015
  
```

图 6-2 逻辑模块目录结构

➤ event: 事件处理模块。

➤ module: 逻辑模块。

### 3. net: 网络模块

游戏的网络部分的包，包括使用的网络框架及网络请求处理等。

➤ rpc: 负责逻辑服务器与其他服务器的远程服务调用。

➤ NetFramework: 模拟网络层框架，负责网络请求响应的封装（这里只写了一个模拟类，正式开发中应当使用 Netty 或 Mina 等网络框架）。

➤ NetHandler: 模拟网络框架处理接口，负责处理网络框架的逻辑。

➤ NetHandlerImpl: 模拟网络框架处理接口的实现类。

➤ NetModule: 模拟网络层框架封装，负责封装网络框架。

➤ ProtoIds: 网络请求协议号，客户端与服务器交互的协议号。

➤ ProtoMessage: 网络请求消息体，客户端与服务器交互的消息结构。

➤ ResultCode: 网络响应码，客户端与服务器交互返回结构的响应码封装。

➤ net.properties: 网络配置 IP 端口文件。

### 4. task: 任务模块

游戏定时任务管理包，包括游戏中需要的定时任务。

➤ JobMgr: Job 管理类。

➤ QuartzManager: Quartz 管理类。

➤ TestJob: 测试 Job 类。

### 5. template: 静态数据

游戏服务器的静态数据包，包括游戏中静态数据的载入和读取。

➤ DataLoader: 游戏服务器数据载入类。

### 6. util: 工具类

工具类包括项目中所需要的工具类，如数据库工具类、缓存工具类、线程池工具类等。

- CacheUtil: 缓存管理类。
- DBUtil: 数据库管理类。
- ExecutorPool: 线程池管理类。

## 7. test

Demo: 游戏服务器逻辑框架测试类（为了更符合逻辑流程，我将其做成了以 GUI 形式来展示的测试 Demo，通过 GUI 界面的输入/输出达到玩家请求模拟的效果），代码如下：

```
package com.hjc.test;

import java.awt.Color;

/**
 * @ClassName: Demo
 * @Description: 服务器模拟测试窗体程序
 * @author 何金成
 * @date 2016年4月27日 下午5:31:18
 *
 */
public class Demo extends JFrame {
    private static final long serialVersionUID = 7163462189849326849L;
    private JPanel contentPane;
    private static JTextArea console;
    private JTextArea reqText;
    private static JTextArea respText;
    private JLabel stateText;
    private static JScrollPane scroll;
    private JButton shutBtn;
    private JButton startBtn;
    private JButton sendBtn;

    public static void print(String msg) {
        System.out.println(msg);
        console.setText(console.getText() + msg + "\r\n");
        // 保持滚动条位于底部
        scroll.getVerticalScrollBar().setValue(
            scroll.getVerticalScrollBar().getMaximum());
    }
}
```

```

    }

    public static void resp(String msg) {
        System.out.println(msg);
        respText.setText(msg);
    }

    /**
     * Launch the application.
     */
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    Demo frame = new Demo();
                    frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

    /**
     * Create the frame.
     */
    public Demo() {
        setResizable(false);
        setTitle("游戏服务器 Demo");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds(100, 100, 751, 710);
        contentPane = new JPanel();
        contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
        setContentPane(contentPane);
        contentPane.setLayout(null);

        startBtn = new JButton("开启服务器");
        startBtn.setFont(new Font("微软雅黑", Font.PLAIN, 12));
        startBtn.addActionListener(new ActionListener() {

```



```
        public void actionPerformed(ActionEvent e) {
            GameInit.start();
            stateText.setText("服务器状态: 开启");
            startBtn.setEnabled(false);
            shutBtn.setEnabled(true);
            sendBtn.setEnabled(true);
        }
    });
    startBtn.setBounds(10, 78, 220, 34);
    contentPane.add(startBtn);

    shutBtn = new JButton("关闭服务器");
    shutBtn.setEnabled(false);
    shutBtn.setFont(new Font("微软雅黑", Font.PLAIN, 12));
    shutBtn.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            GameInit.shut();
            stateText.setText("服务器状态: 关闭");
            startBtn.setEnabled(true);
            shutBtn.setEnabled(false);
            sendBtn.setEnabled(false);
        }
    });
    shutBtn.setBounds(509, 78, 220, 34);
    contentPane.add(shutBtn);

    scroll = new JScrollPane();
    scroll.setBounds(10, 447, 719, 219);
    scroll.setBorder(new LineBorder(new Color(0, 0, 0), 2));
    contentPane.add(scroll);
    // 分别设置水平和垂直滚动条自动出现
    scroll.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_
    SCROLLBAR_AS_NEEDED);
    scroll.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_
    AS_NEEDED);

    console = new JTextArea();
    scroll.setViewportViewView(console);
    console.setFont(new Font("微软雅黑", Font.PLAIN, 14));
```

```

console.setMargin(new Insets(5, 5, 5, 5));
console.setLineWrap(true);
console.setEditable(false);
console.setColumns(100);
console.setBackground(Color.LIGHT_GRAY);
JScrollPane sendScroll = new JScrollPane();
sendScroll.setBounds(10, 165, 719, 106);
contentPane.add(sendScroll);
sendScroll.setBorder(new LineBorder(new Color(0, 0, 0), 2));
// 分别设置水平和垂直滚动条自动出现
sendScroll
    .setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_
SCROLLBAR_AS_NEEDED);
sendScroll
    .setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_
AS_NEEDED);

reqText = new JTextArea();
sendScroll.setViewportViewView(reqText);
reqText.setMargin(new Insets(5, 5, 5, 5));
reqText.setLineWrap(true);
reqText.setFont(new Font("微软雅黑", Font.PLAIN, 14));
reqText.setEditable(true);
reqText.setColumns(100);
reqText.setBackground(Color.WHITE);
reqText.setText("{protoid:0,userid:0,data:{}}");

stateText = new JLabel("服务器状态: 关闭");
stateText.setFont(new Font("微软雅黑", Font.PLAIN, 16));
stateText.setBounds(308, 77, 201, 34);
contentPane.add(stateText);

JLabel lbldemo = new JLabel("游戏服务器测试 Demo");
lbldemo.setFont(new Font("微软雅黑", Font.BOLD, 22));
lbldemo.setBounds(260, 20, 308, 23);
contentPane.add(lbldemo);

JLabel label_1 = new JLabel("控制台: ");
label_1.setFont(new Font("微软雅黑", Font.PLAIN, 14));

```

```
label_1.setBounds(10, 422, 95, 15);
contentPane.add(label_1);

JScrollPane scrollPane = new JScrollPane();
scrollPane

    .setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_A
S_NEEDED);
    scrollPane

        .setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_
SCROLLBAR_AS_NEEDED);
    scrollPane.setBorder(new LineBorder(new Color(0, 0, 0), 2));
    scrollPane.setBounds(10, 305, 719, 106);
    contentPane.add(scrollPane);

respText = new JTextArea();
scrollPane.setViewportViewView(respText);
respText.setFont(new Font("微软雅黑", Font.PLAIN, 14));

respText.setMargin(new Insets(5, 5, 5, 5));
respText.setLineWrap(true);
respText.setEditable(false);
respText.setColumns(100);
respText.setBackground(Color.LIGHT_GRAY);

JButton clearBtn = new JButton("清空控制台");
clearBtn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        console.setText("");
    }
});
clearBtn.setFont(new Font("微软雅黑", Font.PLAIN, 12));
clearBtn.setBounds(91, 419, 139, 23);
contentPane.add(clearBtn);

sendBtn = new JButton("发出请求");
sendBtn.setEnabled(false);
sendBtn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
```



```

        try {
            JSONObject req = JSON.parseObject(reqText.getText());
            // 向网络模块的读取消息队列添加处理消息
            NetFramework.queue.add(req);
            respText.setText("等待响应中...");
        } catch (Exception exception) {
            JOptionPane.showMessageDialog(Demo.this, "请求参数必须是
JSON 格式!");
        }
    }
});
sendBtn.setFont(new Font("微软雅黑", Font.PLAIN, 12));
sendBtn.setBounds(91, 137, 139, 23);
contentPane.add(sendBtn);

JLabel label = new JLabel("响应结果: ");
label.setFont(new Font("微软雅黑", Font.PLAIN, 14));
label.setBounds(10, 280, 235, 15);
contentPane.add(label);

JLabel label_2 = new JLabel("请求数据: ");
label_2.setFont(new Font("微软雅黑", Font.PLAIN, 14));
label_2.setBounds(10, 140, 163, 15);
contentPane.add(label_2);
}
}

```

**【运行结果】**通过 Java 的 AWT 界面模拟游戏服务器客户端的输入/输出及服务器的控制台信息显示, 界面中包括开启服务器、关闭服务器、发出请求和清空控制台的功能。界面整体如图 6-3 所示。

#### (1) 开启服务器。

单击“开启服务器”按钮可开启服务器, 服务器将初始化各部分功能模块。

按钮可用状态及服务器状态显示变化如图 6-4 所示。

控制台输出如图 6-5 所示。



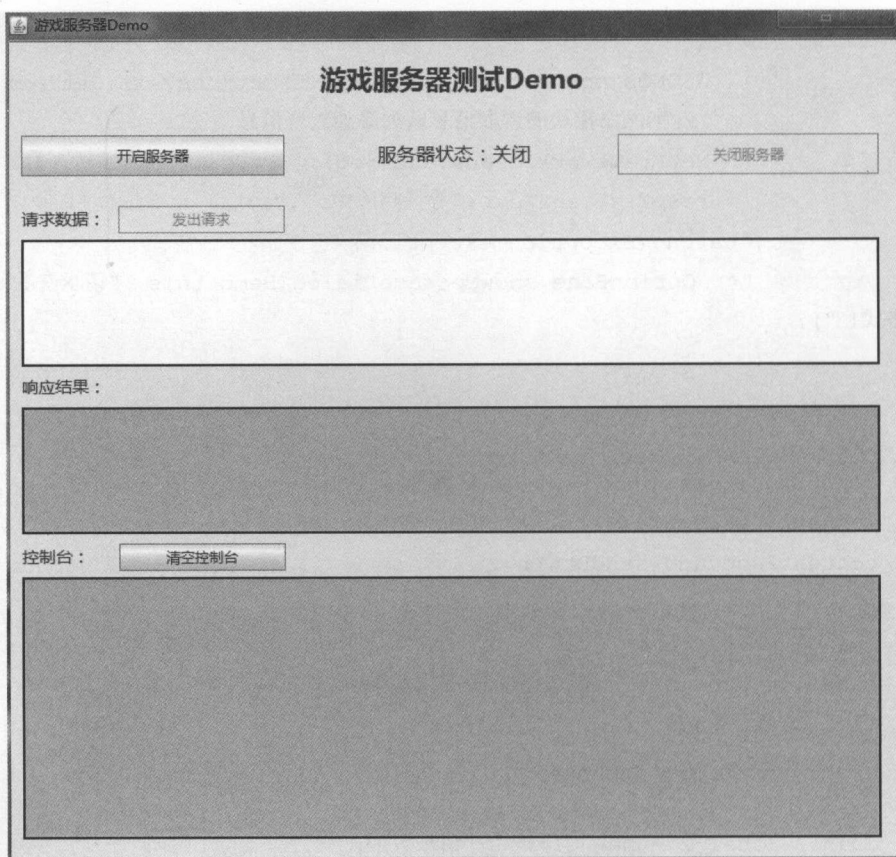


图 6-3 Demo 界面

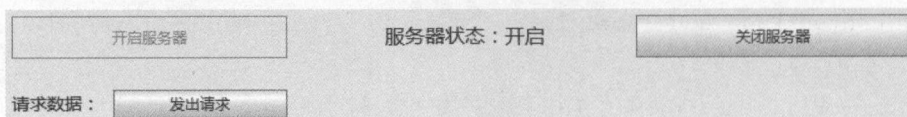


图 6-4 Demo 启动服务器状态改变

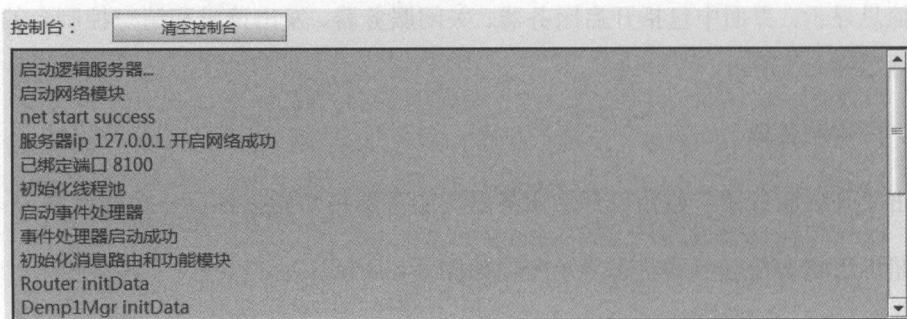


图 6-5 Demo 启动服务器控制台输出

## (2) 发出请求。

在请求数据下方的输入框输入请求数据，再单击“发出请求”按钮，系统便将此请求消息加入到消息处理队列，服务器 Demo 请求队列取出请求消息进行模拟请求处理。响应结果如图 6-6 所示。

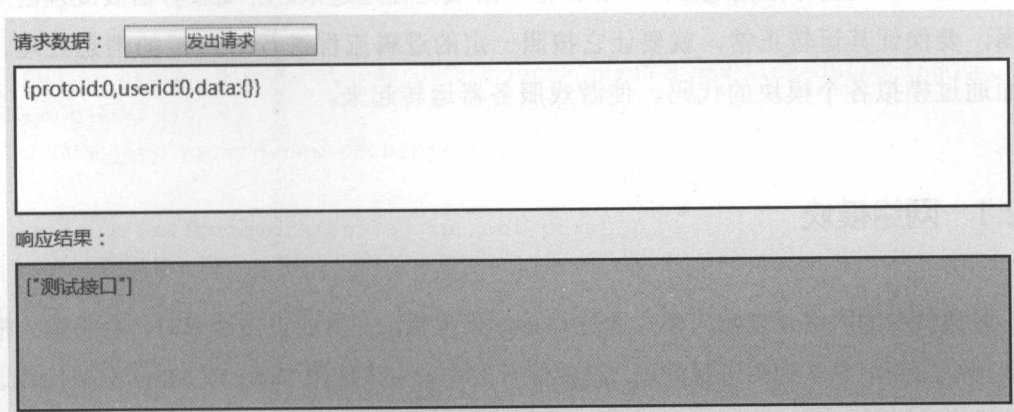


图 6-6 Demo 请求数据

## (3) 关闭服务器。

单击“关闭服务器”按钮，关闭所有相关模块。控制台输出如图 6-7 所示。

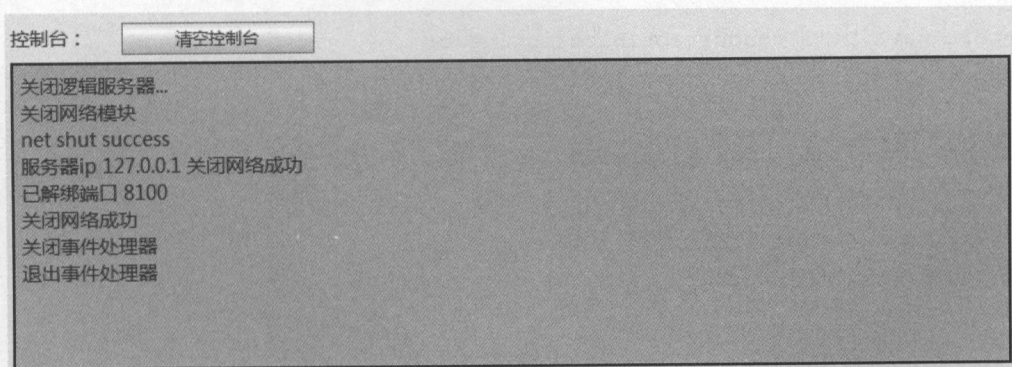


图 6-7 Demo 关闭服务器

**【代码解析】**通过 Java 的 Swing 构建一个模拟服务器界面，并在界面中实现了开启和关闭服务器、发送请求等基本功能。

## 6.2

## 逻辑流程

搭建好游戏整体框架之后,需要让整个游戏流程跑起来。游戏服务器要处理游戏数据,要保证其运转正常,就要让它按照一定的逻辑流程进行客户端的消息处理。下面通过模拟各个模块的代码,使游戏服务器运转起来。

### 6.2.1 网络模块

服务器中的网络请求响应模式通过 Queue 来模拟,在界面点击请求时,会将输入框的数据添加到消息队列中进行处理。在实际开发中,应该使用 Netty 或 Mina 框架代替这部分模拟的网络框架。此模拟部分仅仅是为了模拟本章要讲解的逻辑模块的网络接入部分。以下分别介绍各个类的作用。

**NetFramework:** 模拟网络模块,一般直接使用 Netty 或 Mina 框架。

```
package com.hjc.demo.net;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.hjc.demo.core.GameInit;
import com.hjc.test.Demo;

/**
 * @ClassName: NetFramework
 * @Description: 模拟网络模块,一般直接使用 Netty 或 Mina 框架
 * @author 何金成
 * @date 2016年4月27日 下午3:58:20
 *
 */
public class NetFramework {
    private NetHandler handler;
```

```

private String ip;
private int port;
public static NetFramework inst;

/**
 * @Fields queue : 模拟网络消息处理队列
 */
public static BlockingQueue<JSONObject> queue = new LinkedBlockingQueue
<JSONObject>();
JSONObject exit = new JSONObject();

private NetFramework(String ip, int port) {
    this.setIp(ip);
    this.setPort(port);
    inst = this;
}

public static NetFramework buildNetFramework(String ip, int port) {
    return new NetFramework(ip, port);
}

public void start() throws Exception {
    Demo.print("net start success");
    reading();
}

public void shut() throws Exception {
    Demo.print("net shut success");
    queue.add(exit);
}

public void setHandler(NetHandler handler) {
    this.handler = handler;
}

/**
 * @throws Exception
 * @Title: readMsg
 * @Description: 网络模块消息输入接口
 * @throws

```



```
*/
public void reading() {
    Thread readingThread = new Thread(new Runnable() {
        @Override
        public void run() {
            while (GameInit.state == 1) {
                JSONObject msg = null;
                try {
                    msg = queue.take();
                } catch (InterruptedException e) {
                    Demo.print("Exception when take" + msg);
                    continue;
                }
                if (msg == exit) {
                    break;
                }
                try {
                    if (handler == null) {
                        throw new Exception("handler is not setted");
                    }
                    handler.read(msg);
                } catch (Throwable t) {
                    t.printStackTrace();
                }
            }
            Demo.print("关闭网络模块");
        }
    }, "NetFramework");
    readingThread.start();
}

/**
 * @throws Exception
 * @Title: write
 * @Description: 网络模块消息输出接口
 * @throws
 */
public void write(JSON request) {
    if (handler == null) {
        Demo.print("handler is not setted");
    }
}
```

```

        return;
    }
    handler.write(request);
}

public String getIp() {
    return ip;
}

public void setIp(String ip) {
    this.ip = ip;
}

public int getPort() {
    return port;
}

public void setPort(int port) {
    this.port = port;
}
}

```

**NetHandler:** 网络消息处理接口。

```

package com.hjc.demo.net;

import com.alibaba.fastjson.JSON;

/**
 * @ClassName: NetHandler
 * @Description: 网络消息处理接口
 * @author 何金成
 * @date 2016年4月27日 下午3:31:04
 *
 */
public interface NetHandler {
    public void read(JSON request);

    public void write(JSON request);
}

```

```

public void connect(String host);

public void disconnect(String host);

public void exceptionCaught(String host);

}

```

**NetHandlerImpl:** 网络消息处理实现类。

```

package com.hjc.demo.net;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.hjc.demo.core.Router;
import com.hjc.demo.util.ExecutorPool;
import com.hjc.test.Demo;

/**
 * @ClassName: NetHandler
 * @Description: 网络消息处理实现类
 * @author 何金成
 * @date 2016年4月27日 下午3:31:04
 */
public class NetHandlerImpl implements NetHandler {

    @Override
    public void read(final JSON request) {
        // 使用线程池处理消息。尽快返回处理线程，提高并发效率
        ExecutorPool.handleThreadPool.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    // TODO 请求消息接入逻辑处理
                    Demo.print("read :" + request.toJSONString());
                    Router.getInstance().handle(NetFramework.inst,
                        (JSONObject) request);
                } catch (Exception e) {
                    e.printStackTrace();
                    Demo.print(e.getMessage());
                }
            }
        });
    }
}

```



```

        }

    }

});

}

@Override
public void write(JSON response) {
    Demo.resp(response.toJSONString());
    Demo.print("write:" + response.toJSONString());
}

@Override
public void connect(String host) {

}

@Override
public void disconnect(String host) {

}

@Override
public void exceptionCaught(String host) {

}

}

```

**NetModule:** 网络接入模块，封装于 Mina 或 Netty 之上，进行网络管理。

```

package com.hjc.demo.net;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.nio.charset.Charset;
import java.util.Properties;

import com.hjc.test.Demo;

```



```
/**
 * @ClassName: NetModule
 * @Description: 网络接入模块
 * @author 何金成
 * @date 2016年4月27日 下午3:22:53
 *
 */
public class NetModule {
    private static NetModule netModule = null;
    public static Properties p;
    public static String ip;
    public static int port;
    public static NetFramework net;

    public static NetModule getInstance() {
        if (netModule == null) {
            netModule = new NetModule();
            netModule.init();
        }
        return netModule;
    }

    private NetModule() {

    }

    public void init() {
        try {
            p = readProperties();
            ip = p.getProperty("ip");
            port = Integer.parseInt(p.getProperty("port"));
            net = NetFramework.buildNetFramework(ip, port);
            NetHandler handler = new NetHandlerImpl();
            net.setHandler(handler);
        } catch (IOException e) {
            Demo.print("网络配置文件读取错误");
            e.printStackTrace();
        }
    }
}
```

```

public boolean startNet() {
    // TODO 网络模块启动及绑定端口
    try {
        net.start();
        Demo.print("服务器 ip " + ip + " 开启网络成功");
        Demo.print("已绑定端口 " + port);
    } catch (Exception e) {
        e.printStackTrace();
        Demo.print("服务器 ip " + ip + " 开启网络失败");
    }
    return true;
}

public boolean shutNet() {
    // TODO 网络模块关闭及解绑端口
    try {
        net.shut();
        Demo.print("服务器 ip " + ip + " 关闭网络成功");
        Demo.print("已解绑端口 " + port);
    } catch (Exception e) {
        e.printStackTrace();
        Demo.print("服务器 ip " + ip + " 关闭网络失败");
    }
    return true;
}

protected Properties readProperties() throws IOException {
    Properties p = new Properties();
    InputStream in = NetModule.class.getResourceAsStream("net.
properties");
    Reader r = new InputStreamReader(in, Charset.forName("UTF-8"));
    p.load(r);
    in.close();
    return p;
}
}

```

**【代码解析】**网络模块的代码通过阻塞队列模拟网络消息的读取，网络模块不断循环遍历阻塞队列，当读取到请求消息时，读取消息并交给处理类进行处理。这一模拟过程在实际应用中应换成实际的网络开发框架。

net.properties: 网络的 IP 端口配置文件。

```
port = 8100
ip=127.0.0.1
```

ProtoIds: 存储协议号。

```
package com.hjc.demo.net;

/**
 *
 * @ClassName: ProtoIds
 * @Description: 存储协议号
 * @author 何金成
 * @date 2015 年 5 月 23 日 下午 4:34:41
 */
public class ProtoIds {
    public static final short TEST = 0;
    public static final short DEMO_LOGIC_1 = 1;
    public static final short DEMO_LOGIC_2 = 2;
    public static final short DEMO_LOGIC_3 = 3;
    public static final short DEMO_LOGIC_4 = 4;
    public static final short DEMO_LOGIC_5 = 5;
    public static final short DEMO_LOGIC_6 = 6;
    public static final short DEMO_LOGIC_7 = 7;
    public static final short DEMO_LOGIC_8 = 8;
    public static final short DEMO_LOGIC_9 = 9;
}
```

【代码解析】创建一个存储协议号类，方便对协议号进行集中管理。

ResultCode: 返回码。

```
package com.hjc.demo.net;

/**
 * @ClassName: ResultCode
 * @Description: 返回代码
 * @author 何金成
 * @date 2016 年 5 月 2 日 下午 8:30:29
 */
```



```

*/
public class ResultCode {
    public static final int SUCCESS = 100; // 成功
    public static final int COMMON_ERR = 101; // 默认错误
}

```

【代码解析】网络请求的返回状态码。

## 6.2.2 线程池

游戏中的部分有些需要多线程处理，因此需要线程池管理类管理游戏中的多线程。

ExecutorPool：线程池管理。

```

package com.hjc.demo.util;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * @ClassName: ExecutorPool
 * @Description: 线程池管理
 * @author 何金成
 * @date 2016年4月18日 下午3:42:42
 */
public class ExecutorPool {
    public static ExecutorService handleThreadPool = null;

    public static void initThreadsExecutor() {
        handleThreadPool = Executors.newCachedThreadPool();
    }

    public static void execute(Runnable runnable) {
        handleThreadPool.execute(runnable);
    }

    public static void shutdown() {
        if (!handleThreadPool.isShutdown())
            handleThreadPool.shutdown();
    }
}

```



```

    }
}

```

【代码解析】线程可以单独创建类进行集中管理。

## 6.2.3 启动服务器

单击“启动服务器”按钮，调用以下语句：

```
GameInit.start();
```

GameInit 负责游戏服务器的启动与关闭，代码如下：

```

package com.hjc.demo.core;

import com.hjc.demo.manager.event.EventMgr;
import com.hjc.demo.net.NetModule;
import com.hjc.demo.util.ExecutorPool;
import com.hjc.test.Demo;

/**
 * @ClassName: GameInit
 * @Description: 启动类
 * @author 何金成
 * @date 2016年4月27日 上午11:57:05
 *
 */
public class GameInit {
    public static volatile int state = 0; // 服务器状态：0-关闭，1-开启

    public static boolean start() {
        Demo.print("启动逻辑服务器...");
        GameInit.state = 1;
        Demo.print("启动网络模块");
        NetModule.getInstance().startNet();
        Demo.print("初始化线程池");
        ExecutorPool.initThreadsExecutor();
        Demo.print("启动事件处理器");
        EventMgr.getInstance().init();
        Demo.print("初始化消息路由和功能模块");
    }
}

```

```

Router.getInstance().initData();
Demo.print("开启定时任务调度");
JobMgr.getInstance().initJobs();
return true;
}
}

```

【代码解析】游戏的开启关闭类，此方法包含服务器开启时需要初始化的模块。

## 6.2.4 逻辑请求处理

单击“发出请求”按钮后，调用以下语句：

```

// 向网络模块的读取消息队列添加处理消息
NetFramework.queue.add(req);

```

向网络模块添加一条用户请求的消息，在网络模块的队列中读取这条用户请求：

```

handler.read(msg);

```

在网络处理接口处理信息时，直接将此请求信息导向逻辑模块的核心路由类 Router，调用如下：

```

Router.getInstance().handle(NetFramework.inst, (JSONObject) request);

```

通过以上调用，将网络请求中的消息导入到自定义的逻辑处理核心类 Router 中，再通过 Router 将不同的请求根据协议号分发到不同的模块中进行处理，Router 中的代码如下：

```

package com.hjc.demo.core;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONArray;
import com.alibaba.fastjson.JSONObject;
import com.hjc.demo.manager.module1.Demo1Mgr;
import com.hjc.demo.manager.module2.Demo2Mgr;
import com.hjc.demo.manager.module3.Demo3Mgr;
import com.hjc.demo.manager.module4.Demo4Mgr;
import com.hjc.demo.manager.module5.Demo5Mgr;
import com.hjc.demo.manager.module6.Demo6Mgr;

```

```
import com.hjc.demo.manager.module7.Demo7Mgr;
import com.hjc.demo.manager.module8.Demo8Mgr;
import com.hjc.demo.manager.module9.Demo9Mgr;
import com.hjc.demo.net.NetFramework;
import com.hjc.demo.net.ProtoIds;
import com.hjc.demo.net.ProtoMessage;
import com.hjc.test.Demo;
```

```
/**
```

```
 * @ClassName: Router
```

```
 * @Description: 游戏逻辑处理路由类
```

```
 * @author 何金成
```

```
 * @date 2016年5月2日 下午8:46:16
```

```
 *
```

```
 */
```

```
public class Router {
```

```
    private static Router router;
```

```
    public Demo1Mgr demo1Mgr;
```

```
    public Demo2Mgr demo2Mgr;
```

```
    public Demo3Mgr demo3Mgr;
```

```
    public Demo4Mgr demo4Mgr;
```

```
    public Demo5Mgr demo5Mgr;
```

```
    public Demo6Mgr demo6Mgr;
```

```
    public Demo7Mgr demo7Mgr;
```

```
    public Demo8Mgr demo8Mgr;
```

```
    public Demo9Mgr demo9Mgr;
```

```
    private Router() {
```

```
        demo1Mgr = Demo1Mgr.getInstance();
```

```
        demo2Mgr = Demo2Mgr.getInstance();
```

```
        demo3Mgr = Demo3Mgr.getInstance();
```

```
        demo4Mgr = Demo4Mgr.getInstance();
```

```
        demo5Mgr = Demo5Mgr.getInstance();
```

```
        demo6Mgr = Demo6Mgr.getInstance();
```

```
        demo7Mgr = Demo7Mgr.getInstance();
```

```
        demo8Mgr = Demo8Mgr.getInstance();
```

```
        demo9Mgr = Demo9Mgr.getInstance();
```

```
    }
```

```
    public static Router getInstance() {
```



```

    if (router == null) {
        router = new Router();
    }
    return router;
}

public void initData() {
    Demo.print("Router initData");
    demo1Mgr.initData();
    demo2Mgr.initData();
    demo3Mgr.initData();
    demo4Mgr.initData();
    demo5Mgr.initData();
    demo6Mgr.initData();
    demo7Mgr.initData();
    demo8Mgr.initData();
    demo9Mgr.initData();
}

public void handle(NetFramework net, JSONObject req) throws Exception {
    // 传递数据采用 json 格式, json 中需要包含 ProtoMessage 类中的字段
    if (!req.containsKey("protoid")) {
        NetFramework.inst.write(JSON.parseObject(JSON
            .toString(ProtoMessage.getErrorResp("没有协议号"))));
        return;
    }
    int protoId = req.getIntValue("protoid");
    ProtoMessage reqMsg = JSON.parseObject(req.toString(),
        ProtoMessage.class);
    if (!req.containsKey("userid")) {
        NetFramework.inst.write(JSON.parseObject(JSON
            .toString(ProtoMessage.getErrorResp("没有 userid"))));
        return;
    }
    long userid = reqMsg.getUserid();
    switch (protoId) {
        case ProtoIds.TEST:
            JSONArray ret = new JSONArray();
            ret.add("测试接口");
            NetFramework.inst.write(ret);

```



```

        break;
    case ProtoIds.DEMO_LOGIC_1:
        demo1Mgr.demoLogic1(net, reqMsg, userid);
        break;
    case ProtoIds.DEMO_LOGIC_2:
        demo2Mgr.demoLogic2(net, reqMsg, userid);
        break;
    case ProtoIds.DEMO_LOGIC_3:
        demo3Mgr.demoLogic3(net, reqMsg, userid);
        break;
    case ProtoIds.DEMO_LOGIC_4:
        demo4Mgr.demoLogic4(net, reqMsg, userid);
        break;
    case ProtoIds.DEMO_LOGIC_5:
        demo5Mgr.demoLogic5(net, reqMsg, userid);
        break;
    case ProtoIds.DEMO_LOGIC_6:
        demo6Mgr.demoLogic6(net, reqMsg, userid);
        break;
    case ProtoIds.DEMO_LOGIC_7:
        demo7Mgr.demoLogic7(net, reqMsg, userid);
        break;
    case ProtoIds.DEMO_LOGIC_8:
        demo8Mgr.demoLogic8(net, reqMsg, userid);
        break;
    case ProtoIds.DEMO_LOGIC_9:
        demo9Mgr.demoLogic9(net, reqMsg, userid);
        break;
    default:
        NetFramework.inst.write(JSON.parseObject(JSON
            .toJSONString(ProtoMessage
                .getErrorResp("未注册的协议号" + protoId))));
        break;
    }
}
}
}

```

当消息分发到具体的模块时，不同模块再根据不同的逻辑做不同的处理及不同的响应，如 Demo2Mgr:

```
package com.hjc.demo.manager.module2;
```

```

import com.alibaba.fastjson.JSONArray;
import com.hjc.demo.net.NetFramework;
import com.hjc.demo.net.ProtoMessage;
import com.hjc.test.Demo;

public class Demo2Mgr {
    private static Demo2Mgr demo2Mgr;

    private Demo2Mgr() {

    }

    public static Demo2Mgr getInstance() {
        if (demo2Mgr == null) {
            demo2Mgr = new Demo2Mgr();
        }
        return demo2Mgr;
    }

    public void initData() {
        Demo.print("Demo2Mgr initData");
        // Demo
    }

    public void demoLogic2(NetFramework net, ProtoMessage msg, long userid) {
        Demo.print("invoke demoLogic2");
        JSONArray ret = new JSONArray();
        ret.add(userid);
        ret.add(msg.getData());
        net.write(ret);
    }
}

```

**【运行结果】**模拟一个来自客户端的请求，来请求 Demo2Mgr 的 demoLogic2 方法，得到这个逻辑处理的结果，在界面的请求输入框中输入：

```
{protoid:1,userid:1001,data:{attack:10000,hp:100}}
```

单击“发送请求”按钮，得到的响应及控制台输出如图 6-8 所示。

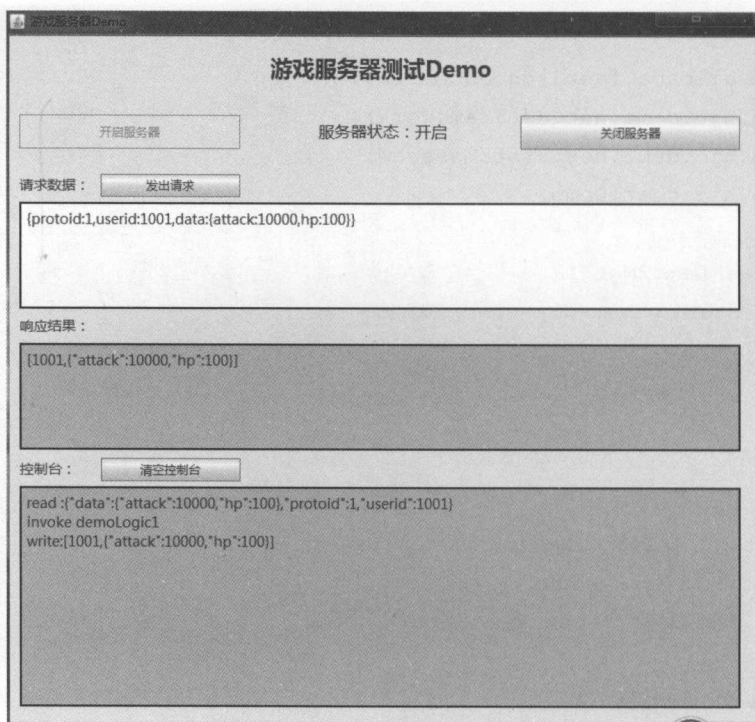


图 6-8 请求 Demo2

由图 6-8 可知模拟客户端得到的响应如下：

```
[1001,{"attack":10000,"hp":100}]
```

客户端得到了服务器的响应之后，便可在前端进行逻辑处理和展示，比如上面示例返回的攻击力及生命值，这些数据就是客户端请求服务器之后返回的数值，可直接展示在玩家基础信息一栏。

### 6.2.5 关闭服务器

单击“关闭服务器”按钮后，调用以下语句：

```
GameInit.shut();
```

即调用 GameInit 中的如下代码：

```
public static boolean shut() {
    Demo.print("关闭逻辑服务器...");
    GameInit.state = 0;
```

```

Demo.print("关闭定时任务调度");
JobMgr.getInstance().stopJobs();
Demo.print("关闭网络模块");
NetModule.getInstance().shutNet();
Demo.print("关闭事件处理器");
EventMgr.shutdown();
return true;
}

```

【代码解析】游戏的开启关闭类，此方法包含服务器关闭时需要关闭释放的模块。

## 6.3 事件处理器

在游戏服务器中，经常会有事件触发机制，触发的事件通常是通过异步方式执行。服务器内部事件处理器，用于断开模块之间耦合。比如登录后要给好友发上线通知，则登录完成后触发一个登录事件，所有关注这个事件的模块，仍然处理自己的业务；避免登录完成后直接调用其他模块的接口。

在事件处理器中，我定义了一个抽象类 EventPro，代码如下：

```

package com.hjc.demo.manager.event;

/**
 * @ClassName: EventProc
 * @Description: 事件处理器
 * @author 何金成
 * @date 2016年5月2日 下午8:58:01
 *
 */
public abstract class EventProc {
    public boolean disable;

    public EventProc() {
        doReg();
    }

    public abstract void proc(Event param);
}

```



```
protected abstract void doReg();
}
```

所有需要实现事件处理的模块需要继承 EventProc 抽象类, 并实现它的 deReg 和 proc 方法, EventPro 代码如下:

```
package com.hjc.demo.manager.event;

/**
 * @ClassName: EventProc
 * @Description: 事件处理器
 * @author 何金成
 * @date 2016年5月2日 下午8:58:01
 */
public abstract class EventProc {
    public boolean disable;

    public EventProc() {
        doReg();
    }

    public abstract void proc(Event param);

    protected abstract void doReg();
}
```

在 GameInit 游戏服务器中, 也会启动一个 EventMgr (事件处理管理类), GameInit 中的 start 方法会调用如下语句:

```
Demo.print("启动事件处理器");
EventMgr.getInstance().init();
```

这个类在一个新的线程中初始化一个阻塞队列, 这个线程不断从阻塞队列中取出需要处理的 Event (事件), 只要有事件添加进去, 就会调用相应的处理类进行处理。

Event 代码如下:

```
package com.hjc.demo.manager.event;

/**
 * @ClassName: Event
```

```

* @Description: 事件
* @author 何金成
* @date 2016 年 5 月 2 日 下午 9:12:30
*
*/
public class Event {
    public int id;
    public Object param;
}

```

EventMgr 代码如下:

```

package com.hjc.demo.manager.event;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

import com.hjc.test.Demo;

/**
 * 服务器内部事件处理器, 用于断开模块之间耦合。比如登录后要给好友发上线通知, 则登录完
 * 成后触发一个登录事件, 关心这个事件的模块, 去处理自己的业务, 避免登录完成后直接调用
 * 其他模块的接口
 *
 * @author 何金成
 */
public class EventMgr implements Runnable {
    public static Map<Integer, List<EventProc>> procs;
    public static BlockingQueue<Event> queue = new LinkedBlockingQueue
<Event>();
    public volatile boolean work;
    public static EventMgr inst;
    private static Event exit = new Event();

    private EventMgr() {
        inst = this;
    }
}

```

```
}

public static EventMgr getInstance() {
    if (inst == null) {
        inst = new EventMgr();
    }
    return inst;
}

public void init() {
    procs = new HashMap<Integer, List<EventProc>>();
    work = true;
    new Thread(this, "EventMgr").start();
    Demo.print("事件处理器启动成功");
}

public static void shutdown() {
    queue.add(exit);
}

public static void addEvent(int id, Object param) {
    Event evt = new Event();
    evt.id = id;
    evt.param = param;
    queue.add(evt);
}

@Override
public void run() {
    while (work) {
        Event evt = null;
        try {
            evt = queue.take();
        } catch (InterruptedException e) {
            Demo.print("Exception when take" + evt);
            continue;
        }
        if (evt == exit) {
            break;
        }
    }
}
```



```

List<EventProc> list = procs.get(evt.id);
if (list == null) {
    Demo.print("该事件 id: " + evt.id + ", 没有事件注册过");
    continue;
}
int cnt = list.size();
for (int i = 0; i < cnt; i++) {
    EventProc proc = list.get(i);
    if (proc.disable) {
        continue;
    }
    try {
        proc.proc(evt);
    } catch (Throwable t) {
        Demo.print("处理事件异常 " + evt.id + " " + evt.param + " proc "
            + proc.getClass().getSimpleName() + "");
        Demo.print("异常内容" + t);
    }
}
}
Demo.print("退出事件处理器 ");
}

public static void regist(int id, EventProc proc) {
    List<EventProc> list = procs.get(id);
    if (list == null) {
        list = new ArrayList<EventProc>(1);
        procs.put(id, list);
    }
    list.add(proc);
}

public static void sendEventFinishMessage() {
}
}

```

在 Demo 代码 Demo6Mgr、Demo7Mgr、Demo8Mgr 及 Demo9Mgr 中实现了事件处理机制。Demo6Mgr 代码如下：

```
package com.hjc.demo.manager.module6;
```



```
import com.alibaba.fastjson.JSONArray;
import com.hjc.demo.manager.event.ED;
import com.hjc.demo.manager.event.Event;
import com.hjc.demo.manager.event.EventMgr;
import com.hjc.demo.manager.event.EventProc;
import com.hjc.demo.net.NetFramework;
import com.hjc.demo.net.ProtoMessage;
import com.hjc.test.Demo;

public class Demo6Mgr extends EventProc {
    private static Demo6Mgr demo6Mgr;

    private Demo6Mgr() {

    }

    public static Demo6Mgr getInstance() {
        if (demo6Mgr == null) {
            demo6Mgr = new Demo6Mgr();
        }
        return demo6Mgr;
    }

    public void initData() {
        Demo.print("Demo6Mgr initData");
        // Demo
    }

    public void demoLogic6(NetFramework net, ProtoMessage msg, long userid) {
        Demo.print("invoke demoLogic6");
        // 触发事件 1
        EventMgr.addEvent(ED.EVE1, userid);
        JSONArray ret = new JSONArray();
        ret.add(userid);
        ret.add(msg.getData());
        net.write(ret);
    }

    @Override
    public void proc(Event param) {
```

```

    if (param.id == ED.EVE1) {
        Long userid = (Long) param.param;
        Demo.print("invoke event1,param:" + userid);
    }
}

@Override
protected void doReg() {
    // 实现注册事件
    EventMgr.regist(ED.EVE1, this);
}
}

```

【运行结果】模拟一个来自客户端的请求，来请求 Demo6Mgr 的 demoLogic6 方法，然后得到这个逻辑处理结果，并异步触发 Demo6Mgr 的事件处理，在界面的请求输入框中输入：

```
{protoid:6,userid:1001,data:{attack:10000,hp:100}}
```

单击“发送请求”按钮，得到的响应及控制台输出如图 6-9 所示。

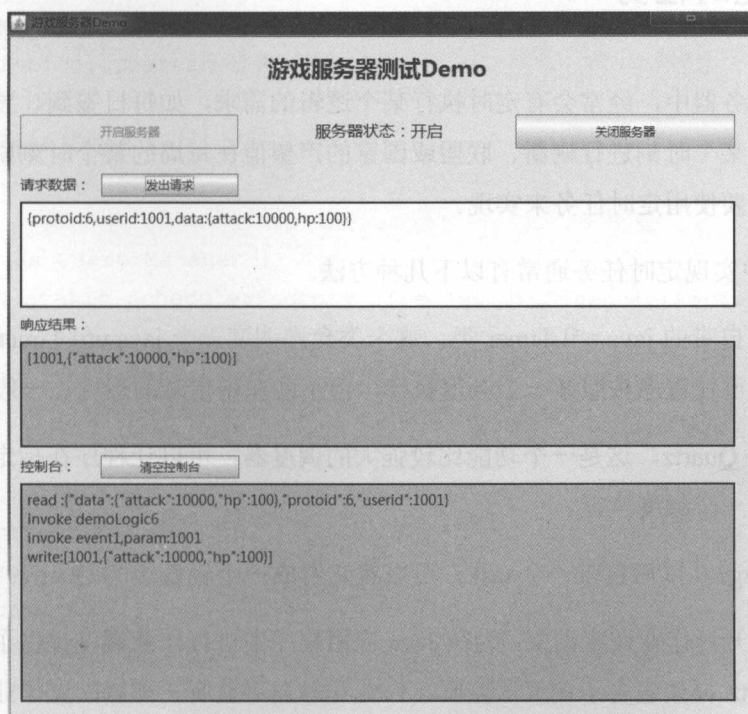


图 6-9 请求 Demo6

如图 6-9 可知模拟客户端得到的响应如下:

```
[1001,{"attack":10000,"hp":100}]
```

控制台输出如下:

```
read :{"data":{"attack":10000,"hp":100},"protoid":6,"userid":1001}  
invoke demoLogic6  
write:[1001,{"attack":10000,"hp":100}]  
invoke event1,param:1001
```

从控制台可以看出,逻辑模块除了正常响应客户端的请求之外,还异步触发了事件处理器,传入了 `userid` 并进行了输出。

**【代码解析】**通过 `EventMgr`、`Event` 和 `EventProc` 类实现了模块之间事件的解耦。在实际项目开发中,会有很多类似需要模块之间通过事件触发来解耦的需求,通常需要自己定义一个事件处理机制来处理相关触发事件。

## 6.4 定时任务

在游戏服务器中,经常会有定时执行某个逻辑的需求,如每日签到、整点更新、玩家某个属性在某个时刻进行刷新、联盟或国家的声望值在每周的某个时刻刷新,这一系列的需求都需要使用定时任务来实现。

在 Java 中实现定时任务通常有以下几种方法。

(1) Java 自带的 `java.util.Timer` 类,这个类允许调度一个 `java.util.TimerTask` 任务。使用这种方式可使程序按照某一个频度执行,但不能在指定时间运行,一般使用较少。

(2) 使用 `Quartz`,这是一个功能比较强大的调度器,可以让程序在指定时间执行,也可以按照某一个频度执行。

(3) `Spring3.0` 以后自带一个 `task`,可以将它看成一个轻量级的 `Quartz`。

`Quartz` 是开源作业调度框架,为在 Java 应用程序中进行作业调度提供了简单却强大的机制。`Quartz` 框架包含了调度器监听、作业和触发器监听。可以配置作业和触发器监听为全局监听或者特定于作业和触发器的监听。`Quartz` 允许开发人员根据时间间隔调度

作业。它实现了作业和触发器的多对多关系，还能把多个作业与不同的触发器关联。整合了 Quartz 的应用程序可以重用来自不同事件的作业，可以为一个事件组合多个作业，可以和 Spring 配置整合使用。Quartz 在功能上远远超越了 JDK 自带的 Timer。

在项目中，我们创建了一个 QuartzManager 类来实现 Quartz 的增、删、改的封装，代码如下：

```
package com.hjc.demo.task;

import java.text.ParseException;
import org.quartz.CronTrigger;
import org.quartz.Job;
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.SchedulerFactory;
import org.quartz.Trigger;
import org.quartz.impl.StdSchedulerFactory;

/**
 * @ClassName: QuartzManager
 * @Description: Quartz 管理类
 * @author 何金成
 * @date 2016年4月28日 下午10:09:47
 *
 */
public class QuartzManager {
    private static SchedulerFactory sf = new StdSchedulerFactory();
    private static String JOB_GROUP_NAME = "group1";
    private static String TRIGGER_GROUP_NAME = "trigger1";

    /**
     * @Title: addJob
     * @Description: 添加 Job
     * @param jobName
     * @param job
     * @param time
     * @throws SchedulerException
     * @throws ParseException
     * @return void
     */
}
```



```

    * @throws
    */
    public static void addJob(String jobName, Job job, String time)
        throws SchedulerException, ParseException {
        Scheduler sched = sf.getScheduler();
        JobDetail jobDetail = new JobDetail(jobName, JOB_GROUP_NAME,
            job.getClass()); // 任务名、任务组、任务执行类
        // 触发器
        CronTrigger trigger = new CronTrigger(jobName, TRIGGER_GROUP_NAME);
        // 触发器名、触发器组
        trigger.setCronExpression(time); // 触发器时间设定
        sched.scheduleJob(jobDetail, trigger);
        // 启动
        if (!sched.isShutdown())
            sched.start();
    }

    /**
     * @Title: addJob
     * @Description: 添加 Job
     * @param jobName
     * @param jobGroupName
     * @param triggerName
     * @param triggerGroupName
     * @param job
     * @param time
     * @throws SchedulerException
     * @throws ParseException
     * @return void
     * @throws
     */
    public static void addJob(String jobName, String jobGroupName,
        String triggerName, String triggerGroupName, Job job, String time)
        throws SchedulerException, ParseException {
        Scheduler sched = sf.getScheduler();
        JobDetail jobDetail = new JobDetail(jobName, jobGroupName,
            job.getClass()); // 任务名、任务组、任务执行类
        // 触发器
        CronTrigger trigger = new CronTrigger(triggerName, triggerGroupName);
        // 触发器名、触发器组

```

```

        trigger.setCronExpression(time); // 触发器时间设定
        sched.scheduleJob(jobDetail, trigger);
        if (!sched.isShutdown())
            sched.start();
    }

    /**
     * @Title: modifyJobTime
     * @Description: 修改 Job 触发时间
     * @param jobName
     * @param time
     * @throws SchedulerException
     * @throws ParseException
     * @return void
     * @throws
     */
    public static void modifyJobTime(String jobName, String time)
        throws SchedulerException, ParseException {
        Scheduler sched = sf.getScheduler();
        Trigger trigger = sched.getTrigger(jobName, TRIGGER_GROUP_NAME);
        if (trigger != null) {
            CronTrigger ct = (CronTrigger) trigger;
            ct.setCronExpression(time);
            sched.resumeTrigger(jobName, TRIGGER_GROUP_NAME);
        }
    }

    /**
     * @Title: modifyJobTime
     * @Description: 修改任务触发时间
     * @param triggerName
     * @param triggerGroupName
     * @param time
     * @throws SchedulerException
     * @throws ParseException
     * @return void
     * @throws
     */
    public static void modifyJobTime(String triggerName,
        String triggerGroupName, String time) throws SchedulerException,

```

```

        ParseException {
            Scheduler sched = sf.getScheduler();
            Trigger trigger = sched.getTrigger(triggerName, triggerGroupName);
            if (trigger != null) {
                CronTrigger ct = (CronTrigger) trigger;
                // 修改时间
                ct.setCronExpression(time);
                // 重启触发器
                sched.resumeTrigger(triggerName, triggerGroupName);
            }
        }

/**
 * @Title: removeJob
 * @Description: 移除任务
 * @param jobName
 * @throws SchedulerException
 * @return void
 * @throws
 */
public static void removeJob(String jobName) throws SchedulerException {
    sched = sf.getScheduler();
    sched.pauseTrigger(jobName, TRIGGER_GROUP_NAME); // 停止触发器
    sched.unscheduleJob(jobName, TRIGGER_GROUP_NAME); // 移除触发器
    sched.deleteJob(jobName, JOB_GROUP_NAME); // 删除任务
}

/**
 * @Title: removeJob
 * @Description: 移除任务
 * @param jobName
 * @param jobGroupName
 * @param triggerName
 * @param triggerGroupName
 * @throws SchedulerException
 * @return void
 * @throws
 */
public static void removeJob(String jobName, String jobGroupName,
                             String triggerName, String triggerGroupName)

```



```

        throws SchedulerException {
    Scheduler sched = sf.getScheduler();
    sched.pauseTrigger(triggerName, triggerGroupName); // 停止触发器
    sched.unscheduleJob(triggerName, triggerGroupName); // 移除触发器
    sched.deleteJob(jobName, jobGroupName); // 删除任务
    }
}

```

我们把需要执行的定时任务继承自 `org.quartz.Job`，创建 `TestJob`，代码如下：

```

package com.hjc.demo.task;

import java.util.Date;

import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

import com.hjc.test.Demo;

/**
 * @ClassName: TestJob
 * @Description: 测试 Job
 * @author 何金成
 * @date 2016年4月28日 下午10:16:04
 */
public class TestJob implements Job {

    @Override
    public void execute(JobExecutionContext ctx) throws JobExecutionException {
        Demo.print("现在时间: " + new Date().toLocaleString());
    }
}

```

当定时任务很多时，便需要一个定时任务管理类，来管理它们的启动和关闭，创建 `JobMgr`，代码如下：

```

package com.hjc.demo.task;

```



```
import java.text.ParseException;

import org.quartz.SchedulerException;

import com.hjc.test.Demo;

/**
 * @ClassName: JobMgr
 * @Description: Job 管理类
 * @author 何金成
 * @date 2016年4月28日 下午10:23:51
 *
 */
public class JobMgr {
    private static JobMgr jobMgr;
    private TestJob job = new TestJob();

    private JobMgr() {

    }

    public static JobMgr getInstance() {
        if (jobMgr == null) {
            jobMgr = new JobMgr();
        }
        return jobMgr;
    }

    /**
     * @Title: initJobs
     * @Description: 初始化所有任务
     * @return void
     * @throws
     */
    public void initJobs() {
        try {
            // 添加 Job 定时任务
            QuartzManager.addJob(TestJob.class.getSimpleName(), job,
                "0/2 * * * * ?");
        } catch (SchedulerException | ParseException e) {
```

```

        e.printStackTrace();
    }
    Demo.print("开启定时任务成功");
}

/**
 * @Title: stopJobs
 * @Description: 关闭所有任务
 * @return void
 * @throws
 */
public void stopJobs() {
    try {
        QuartzManager.removeJob(TestJob.class.getSimpleName());
    } catch (SchedulerException e) {
        e.printStackTrace();
    }
}
}

```

在 GameInit 启动游戏服务器时,需要调用 JobMgr 的 initJobs 来初始化所有定时任务,并且在服务器关闭时调用 stopJobs 移除所有的定时任务。

开启定时任务调度的语句如下:

```

Demo.print("开启定时任务调度");
JobMgr.getInstance().initJobs();

```

关闭定时任务调度的语句如下:

```

Demo.print("关闭定时任务调度");
JobMgr.getInstance().stopJobs();

```

**【运行结果】**启动游戏服务器时,调用 JobMgr 的 initJobs,并初始化 TestJob,TestJob 完成的内容是每两秒调用一次执行逻辑,执行逻辑中进行了输出。添加了定时任务时,启动服务器的控制台输出如图 6-10 所示。

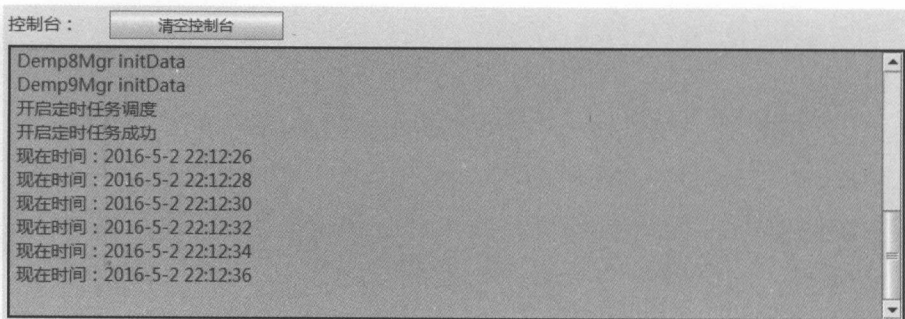


图 6-10 定时任务调度

**【代码解析】**通过 ScheduleFactory 获取 Schedule 对象，给 Schedule 对象设置好执行计划 ConTrigger 与计划任务 Job，就可以启用定时任务。通过 QuartzManager 对计划任务进行封装，使用时只需要使类继承自 Job 类，然后在 JobMgr 中添加任务即可。

## 6.5 RPC 框架

很多时候，在游戏服务器中所有的逻辑不是在一台服务器上完成，一个逻辑也可能均分到多个服务器进行，一个好的服务器架构，应该根据需求将游戏服务器划分为多个模块，每个模块完成特定的需求。比如游戏中的支付功能划分为一个支付服务器，玩家匹配划分为一个匹配服务器，国战划分为一个国战服务器等，不同的服务器之间共同组成游戏服务器的逻辑功能。游戏服务器的架构设计，在后面章节中会讲到，这节主要讲解各个服务器进程之间的通信部分，在多个服务器进程之间的通信，目前使用的技术一般是 RPC（Remote Procedure Call Protocol，远程过程调用协议）。

使用 RPC 可以访问远程主机的进程服务，不需要清楚底层网络通信机制，只需要关注服务本身即可。RPC 是目前分布式开发中一种常用的技术，其在分布式开发中能更简单方便地调用远程服务，使其像本地开发一样。目前，RPC 有很多成熟的框架，如 dubbo、Thrift、Hessian 等。本节主要讲解两款轻量级的 RPC 框架——Json-rpc 和 Motan。

### 6.5.1 Json-rpc

Json-rpc 是远程调用服务，底层使用 Json 作为 RPC 的传输协议，Json-rpc 可以运行

在不同操作系统及不同环境的程序中，只要程序在使用中满足过程调用规范即可。这种远程过程调用可以使用 HTTP 作为传输协议，也可以使用其他传输协议，传输的内容是 Json 消息体。

除了 Json-rpc 之外，还有另一个 RPC 框架——Xml-rpc。Xml-rpc 使用 XML 作为 RPC 数据交互格式，因此它传输的数据不仅格式复杂、体积偏大，更是占用传输的带宽，因此个人建议使用 Json-rpc。在 XML 的解析上，服务器也会耗费更多的资源。相比之下，Json 格式更小巧轻便，解析也非常容易。

Json-rpc 属于轻量级的 RPC 框架，因此使用 Json-rpc 时，实现起来也非常简单，不需要太多的代码，就可以完成本地从远程调用服务的过程。目前 Json-rpc 有 Python、Ruby、.Net、PHP、C、C++、C#、JavaScript、Erlang、Objective-C、Java 等多种语言的实现，并且语言之间通过 Json-rpc 通信也是毫无压力。

其官网地址是：<http://www.jsonrpc.org/>。

其 Wiki 地址是：<http://json-rpc.org/wiki/implementations>。

通过 Wiki 可以看到，有 Python、Ruby、.Net、PHP、C、C++、C#、JavaScript、Erlang、Objective-C、Java 等多种语言的实现，本书以 Java 开发为主题，因此我简单介绍一下 Java 使用 Json-rpc 的示例。

下载 jsonrpc4j 的 jar 包及其依赖的 jar 包，我直接搭建 Maven 项目，使用 Maven 来管理 jar 包，其 pom 文件如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.hjc.demo</groupId>
    <artifactId>jsonrpc_server</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <name>jsonrpc_server Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <!-- jsonrpc4j -->
        <dependency>
```



```
<groupId>com.github.briandilley.jsonrpc4j</groupId>
<artifactId>jsonrpc4j</artifactId>
<version>1.0</version>
</dependency>
<!-- jsonrpc4j 依赖的包 -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.10</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.0.2</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.0.2</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-annotations</artifactId>
  <version>2.0.2</version>
</dependency>
<dependency>
  <groupId>javax.portlet</groupId>
  <artifactId>portlet-api</artifactId>
  <version>2.0</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>3.1.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>3.1.2.RELEASE</version>
</dependency>
```

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>3.1.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>3.1.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>commons-codec</groupId>
  <artifactId>commons-codec</artifactId>
  <version>1.4</version>
</dependency>
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpcore-nio</artifactId>
  <version>4.2.1</version>
</dependency>
<dependency>
  <groupId>org.jmock</groupId>
  <artifactId>jmock-junit4</artifactId>
  <version>2.5.1</version>
</dependency>
<dependency>
  <groupId>org.jmock</groupId>
  <artifactId>jmock</artifactId>
  <version>2.5.1</version>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-server</artifactId>
  <version>9.0.0.RC0</version>
</dependency>
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-servlet</artifactId>
  <version>9.0.0.RC0</version>
</dependency>
```

```

    <dependency>
      <groupId>org.ow2.chameleon.fuchsia.base.json-rpc</groupId>
      <artifactId>org.ow2.chameleon.fuchsia.base.json-rpc.json-rpc-
bundle</artifactId>
      <version>0.0.2</version>
    </dependency>

  </dependencies>
  <build>
    <finalName>jsonrpc_server</finalName>
  </build></project>

```

以上 jar 包等环境部署好之后, 就可以进行开发了, 我们需要部署一个 Servlet, 作为 RPC 调用的接口。(我的环境是 Tomcat6, 环境是 Tomcat7 的朋友可以直接使用 @WebServlet 来部署 Servlet, 具体代码因环境而异)

web.xml:

```

<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>RpcServer</servlet-name>
    <display-name>RpcServer</display-name>
    <description></description>
    <servlet-class>com.hjc.demo.RpcServer</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>RpcServer</servlet-name>
    <url-pattern>/rpc</url-pattern>
  </servlet-mapping></web-app>

```

RPC 服务端的实现:

```

package com.hjc.demo;
import java.io.IOException;
import javax.servlet.ServletException; import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest; import javax.servlet.http.
HttpServletResponse;

```



```
import com.googlecode.jsonrpc4j.JsonRpcServer;

public class RpcServer extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private JsonRpcServer rpcServer = null;

    public RpcServer() {
        super();
        rpcServer = new JsonRpcServer(new DemoServiceImple(), DemoService.class);
    }

    @Override
    protected void service(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        rpcServer.handle(request, response);
    }
}
```

其中，`DemoService` 是 RPC 调用的接口，`DemoServiceImple` 是 `DemoService` 接口的实现，它们的代码分别如下。

#### RPC 调用接口：

```
package com.hjc.demo;

public interface DemoService {

    public DemoBean getDemo(String code, String msg);

    public Integer getInt(Integer code);

    public String getString(String msg);

    public void doSomething();
}
```

#### DemoServiceImple:

```
package com.hjc.demo;

public class DemoServiceImple implements DemoService {

    public DemoBean getDemo(String code, String msg) {
```



```
        DemoBean bean1 = new DemoBean();
        bean1.setCode(Integer.parseInt(code));
        bean1.setMsg(msg);
        return bean1;
    }

    public Integer getInt(Integer code) {
        return code;
    }

    public String getString(String msg) {
        return msg;
    }

    public void doSomething() {
        System.out.println("do something");
    }
}
```

**DemoBean** 是一个普通的 **JavaBean**，其代码如下：

```
package com.hjc.demo;
import java.io.Serializable;
public class DemoBean implements Serializable{
    private static final long serialVersionUID = -5141784402935371524L;
    private int code;
    private String msg;

    public int getCode() {
        return code;
    }

    public void setCode(int code) {
        this.code = code;
    }

    public String getMsg() {
        return msg;
    }
}
```

```

public void setMsg(String msg) {
    this.msg = msg;
}
}

```

完成了以上代码即完成了服务端的代码，下面是一个客户端的测试类。

**JsonRpc 客户端测试类：**

```

package com.hjc.test;
import java.net.MalformedURLException;
import java.net.URL;import java. util.ArrayList;
import java.util.HashMap;import java.util.List;
import java. util.Map;
import com.fasterxml.jackson.databind.node.ObjectNode;
import com.googlecode. jsonrpc4j.JsonRpcHttpClient;
import com.hjc.demo.DemoBean;
public class JsonRpcTest {
    static JsonRpcHttpClient client;

    public JsonRpcTest() {

    }

    public static void main(String[] args) throws Throwable {
        // 实例化请求地址，注意服务端 web.xml 中地址的配置
        try {
            client = new JsonRpcHttpClient(new URL(
                "http://127.0.0.1:8080/jsonrpc_server/rpc"));
            // 请求头中添加的信息
            Map<String, String> headers = new HashMap<String, String>();
            headers.put("UserKey", "hjckey");
            // 添加到请求头中去
            client.setHeaders(headers);
            JsonRpcTest test = new JsonRpcTest();
            test.doSomething();
            DemoBean demo = test.getDemo(1, "哈");
            int code = test.getInt(2);
            String msg = test.getString("哈哈哈哈哈");
            // print
            System.out.println("=====javabean");

```

```

        System.out.println(demo.getCode());
        System.out.println(demo.getMsg());
        System.out.println("=====Integer");
        System.out.println(code);
        System.out.println("=====String");
        System.out.println(msg);
        System.out.println("=====end");
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void doSomething() throws Throwable {
    client.invoke("doSomething", null);
}

public DemoBean getDemo(int code, String msg) throws Throwable {
    String[] params = new String[] { String.valueOf(code), msg };
    DemoBean demo = null;
    demo = client.invoke("getDemo", params, DemoBean.class);
    return demo;
}

public int getInt(int code) throws Throwable {
    Integer[] codes = new Integer[] { code };
    return client.invoke("getInt", codes, Integer.class);
}

public String getString(String msg) throws Throwable {
    String[] msgs = new String[] { msg };
    return client.invoke("getString", msgs, String.class);
}
}

```

【运行结果】最后的打印结果如下。

服务端：

```
do something
```

客户端:

```
=====javabean1
哈
=====Integer2
=====String
哈哈哈
=====end
```

以上是 JsonRpc 在 Java 中应用的 Demo 示例, 它的短连接及 Json 传输特点, 很适合一些轻量级的 Rpc 调用。

**【代码解析】**JsonRpc 的服务端实现通过创建 JsonRpcServer 绑定处理类, 并在 Http 的处理方法中调用 handle 方法, 即可将 JsonRpc 客户端的请求导入到自己的处理类中, 处理完成并返回信息。JsonRpc 的客户端通过创建 JsonRpcHttpClient 对象, 并调用 invoke 方法即可操作 JsonRpc, 其中传入你需要调用方法的传入参数。

## 6.5.2 Motan

Motan 是新浪微博开源的 RPC 轻量级框架, 在 2014 年的春晚中有千亿次的调用, 对抗了春晚的最高峰值。

### 1. 认识 Motan

2013 年, 微博 RPC 框架 Motan 在前辈大师们 (福林、fishermen、小麦、王喆等) 的精心设计和辛勤工作中诞生, 向各位大师们致敬, Motan 也得到了微博各个技术团队的鼎力支持及不断完善, 如今 Motan 在微博平台中已经被广泛应用, 每天为数百个服务完成近千亿次的调用。

——张雷

通过 Motan 在 Github 开源的 Wiki, 可以更加详细地了解到 Motan 的设计思想、使用场景、功能和特色等。

Motan RPC, 作为新浪微博的 RPC 框架, 其底层网络通信使用了 Netty 网络框架, 序列化协议支持 Hessian 和 Java 的序列化, 网络通信协议可以支持 Motan、HTTP、TCP 等, Motan 框架在新浪内部大量使用, 因此在系统的健壮性和服务治理方面, Motan 有较成熟的技术解决方案。Motan 配置管理服务通过 Config 实现了 High Availability 和 Load



Balance 策略。Load Balance 支持灵活的 FailOver、FailFast HA、Round Robin、LRU、Consistent Hashdeng 等策略,因此 Motan 在健壮性上是可观的。而在服务治理方面,Motan 也能生成完整的服务调用链数据、服务请求性能数据、响应时间 (Response Time)、QPS 及标准化 Error、Exception 日志信息。

Motan 提供了实用的服务治理功能和优秀的 RPC 扩展能力。

Motan 提供的主要功能包括:

(1) 发现服务、订阅服务和通知。

(2) 支持 FailOver、FailFast, 以及 Server 连续失败的次数达到规定次数进行心跳检测等高可用策略。

(3) 支持优先低并发、一致性 Hash、随机请求、轮询等负载均衡策略。

(4) 支持 SPI 扩展。

(5) 可调用统计日志和访问日志。

Motan 可以支持不同的 RPC、传输协议。Motan 能够无缝支持 Spring 配置方式使用 RPC 服务,通过简单、灵活的配置就可以提供或使用 RPC 服务。通过使用 Motan 框架,可以十分方便地进行服务拆分、分布式服务部署。

关于 Motan 的更多内容可参考 Motan 的 github 源码: <https://github.com/weibocom/motan>。

## 2. 简单实现

按照 Github 中的 Wiki 介绍创建 maven 项目 motandemo, 并实现一个简单的 Demo。

(1) 添加 pom 依赖。

pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.hjc.demo</groupId>
  <artifactId>motandemo</artifactId>
```

```

<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>
<name>motandemo</name>
<repositories>
  <repository>
    <id>spy</id>
    <name>36</name>
    <layout>default</layout>
    <url>http://repo1.maven.org/maven2</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.weibo</groupId>
    <artifactId>motan-core</artifactId>
    <version>0.0.1</version>
  </dependency>
  <dependency>
    <groupId>com.weibo</groupId>
    <artifactId>motan-transport-netty</artifactId>
    <version>0.0.1</version>
  </dependency>
  <!-- 集群相关 -->
  <dependency>
    <groupId>com.weibo</groupId>
    <artifactId>motan-registry-consul</artifactId>
    <version>0.0.1</version>
  </dependency>
  <dependency>
    <groupId>com.weibo</groupId>
    <artifactId>motan-registry-zookeeper</artifactId>

```

```

        <version>0.0.1</version>
    </dependency>
    <!-- dependencies blow were only needed for spring-based features -->
    <dependency>
        <groupId>com.weibo</groupId>
        <artifactId>motan-springsupport</artifactId>
        <version>0.0.1</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.0.5.RELEASE</version>
    </dependency>
</dependencies>
<build>
    <finalName>motan</finalName>
</build></project>

```

注意：如果 maven 下载不成功可以去仓库直接搜索，再下载 jar 包。仓库地址为 <http://mvnrepository.com/>。

下载完成后使用 maven 进行本地安装，安装命令如下：

```

mvn install:install-file -Dfile=<path-to-file> -DgroupId=<group-id>
-DartifactId=<artifact-id> -Dversion=<version> -Dpackaging=<packaging>

```

确定 pom.xml 不报错之后再进行后续操作。

(2) 为服务方和调用方创建接口。

服务方和调用方都使用接口 FooService:

```

package com.hjc.motan.server;
import java.util.List;import java.util.Map;
import com.hjc.motan.DemoBean;
public interface FooService {
    public String hello(String name);

    public int helloInt(int number1);

    public double helloDouble(double number2);

```

```

public List<String> helloList(List<String> list);

public Map<String, List<String>> helloMap(Map<String, List<String>> map);

public DemoBean helloJavabean(DemoBean bean);
}

```

服务方实现这个接口的逻辑。

### FooServiceImpl:

```

package com.hjc.motan.server;
import java.util.List;import java.util.Map;
import org.springframework.context.ApplicationContext;import org.
springframework. context.support.ClassPathXmlApplicationContext;
import com.hjc.motan.DemoBean;
public class FooServiceImpl implements FooService {

    public static void main(String[] args) throws InterruptedException {
        ApplicationContext applicationContext = new ClassPathXmlApplicationContext(
            "classpath:motan_server.xml");
        System.out.println("server start...");
    }

    public String hello(String name) {
        System.out.println("invoked rpc service " + name);
        return "hello " + name;
    }

    public int helloInt(int number1) {
        System.out.println("invoked rpc service " + number1);
        return number1;
    }

    public double helloDouble(double number2) {
        System.out.println("invoked rpc service " + number2);
        return number2;
    }

    public List<String> helloList(List<String> list) {
        System.out.print("invoked rpc service ");
    }
}

```



```

        for (String string : list) {
            System.out.print(string + ",");
        }
        System.out.println();
        return list;
    }

    public Map<String, List<String>> helloMap(Map<String, List<String>> map) {
        System.out.print("invoked rpc service ");
        for (String key : map.keySet()) {
            System.out.print(key + ":[");
            for (String list : map.get(key)) {
                System.out.print(list + ",");
            }
            System.out.print("],");
        }
        System.out.println();
        return map;
    }

    public DemoBean helloJavabean(DemoBean bean) {
        System.out.print("invoked rpc service " + bean);
        System.out.print(", " + bean.getId());
        System.out.print(", " + bean.getName());
        System.out.print(", " + bean.getScore());
        System.out.println();
        return bean;
    }
}

```

### (3) 配置服务方暴露接口。

在项目根目录 (src/main/datasource) 创建 motan\_server.xml:

```

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.
org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:motan="http://api.weibo.com/schema/motan"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http:// www.springframework.org/schema/beans/spring-beans-2.5.xsd

```

```

http://api.weibo.com/schema/motan http://api.weibo.com/schema/motan.xsd">

<!-- service implementation bean -->
<bean id="serviceImpl" class="com.hjc.motan.server.FooServiceImpl" />
<!-- exporting service by motan -->
<motan:service interface="com.hjc.motan.server.FooService" ref="serviceImpl"
export="8002" /></beans>

```

如果发现 Eclipse 不能自动下载 motan.xsd，就要手动配置。先从 motan-core 的 jar 包中找到 schema 文件，并将其复制到任意位置。然后在 Eclipse 中选择“Window->Preferences->XML->XML Catalog->User Specified Entries”选项，再单击“Add”按钮，输入 Location 和 Key，如图 6-11 所示。

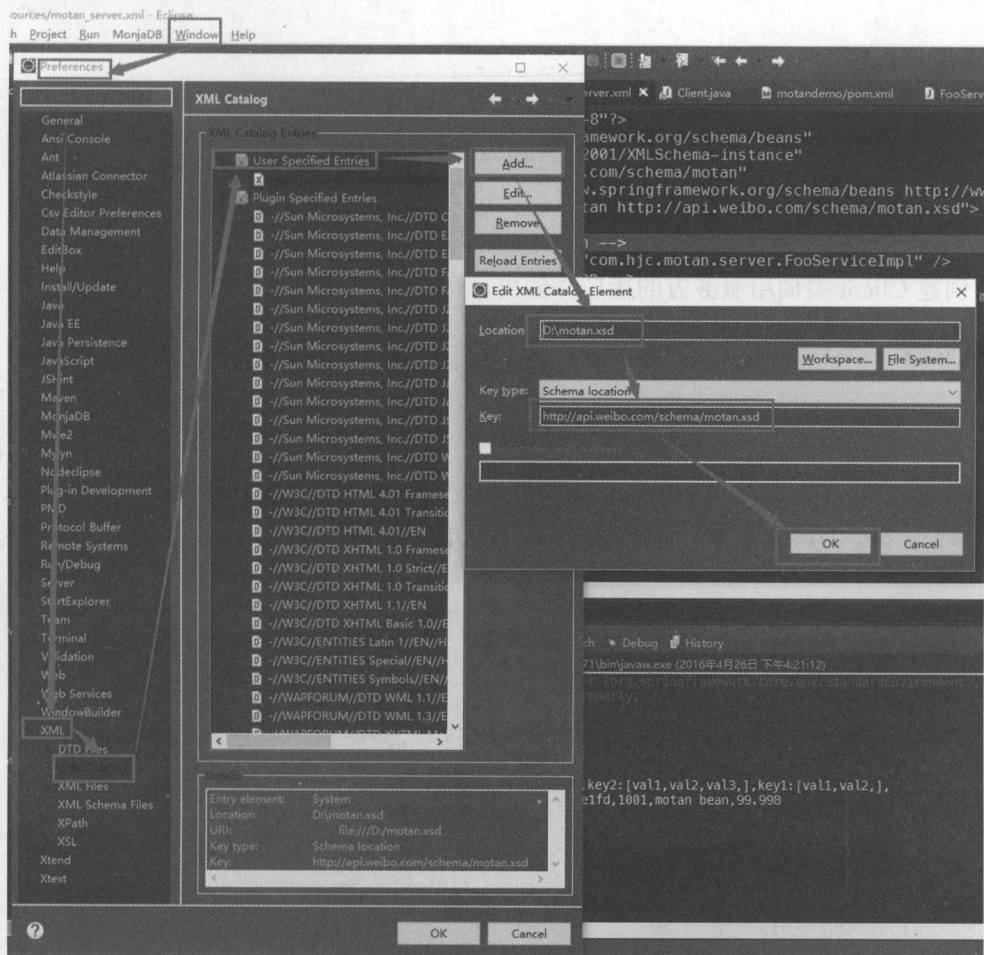


图 6-11 手动添加 schema

执行完上述步骤之后，就可以启动 Motan RPC 的服务方了，在 FooServiceImpl 中已经写好了 main 方法，单击右键运行即可。

#### (4) 配置调用方调用接口。

在项目根目录（src/main/datasource）创建 motan\_server.xml:

```
<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:motan="http://api.weibo.com/schema/motan" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd http://api.weibo.com/schema/motan http://api.weibo.com/schema/motan.xsd">

    <!-- reference to the remote service -->
    <motan:referer id="remoteService" interface="com.hjc.motan.server.FooService" directUrl="localhost:8002"/></beans>
```

#### (5) 调用方调用。

创建 Client 类调用服务方的接口并输出。

Client:

```
package com.hjc.motan.client;
import java.util.Arrays;import java.util.HashMap;import java.util.List;
import java.util.ArrayList;import java.util.Map;
import org.springframework.context.ApplicationContext;import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.hjc.motan.DemoBean;import com.hjc.motan.server.FooService;
public class Client {

    public static void main(String[] args) throws InterruptedException {
        ApplicationContext ctx = new ClassPathXmlApplicationContext(
            "classpath:motan_client.xml");
        // 获取到 service
        FooService service = (FooService) ctx.getBean("remoteService");
        // rpc 调用
        /** String **/
        String ret1 = service.hello("motan");
        System.out.println(ret1);
    }
}
```



```

/** int */
int ret2 = service.helloInt(110);
System.out.println(ret2);
/** double */
double ret3 = service.helloDouble(11.2);
System.out.println(ret3);
/** list */
List<String> list = new ArrayList<String>();
list.add("hello");
list.add("motan");
List<String> ret4 = service.helloList(list);
for (String string : ret4) {
    System.out.print(string + ",");
}
System.out.println();
/** map */
Map<String, List<String>> map = new HashMap<String, List<String>>();
map.put("key1", Arrays.asList(new String[] { "val1", "val2" }));
map.put("key2", Arrays.asList(new String[] { "val1", "val2", "val3" }));
map.put("key3", Arrays.asList(new String[] { "val1", "val2", "val3",
"val4" }));
Map<String, List<String>> ret5 = service.helloMap(map);
for (String key : ret5.keySet()) {
    System.out.print(key + ":[");
    for (String tmp : map.get(key)) {
        System.out.print(tmp + ",");
    }
    System.out.print("],");
}
System.out.println();
/** javabean */
DemoBean bean = new DemoBean();
bean.setId(10011);
bean.setName("motan bean");
bean.setScore(99.998);
DemoBean ret6 = service.helloJavabean(bean);
System.out.print(ret6.getId());
System.out.print(", " + ret6.getName());
System.out.print(", " + ret6.getScore());
System.out.println();

```



```

    }
}

```

启动 Client 的 main 方法开始调用。

【输出结果】通过以上 Demo 创建了 Server 端和 Client 端，分别启动服务方和调用方之后，查看控制台输出如下。

服务方：

```

server start...invoked rpc service motaninvoked rpc service 110invoked rpc
service 11.2invoked rpc service hello,motan,invoked rpc service
key3:[val1,val2,val3,val4,],key2:[val1,val2,val3,],key1:[val1,val2,],in
voked rpc service com.hjc.motan.DemoBean@2cf3elfd,1001,motan bean,99.998

```

调用方：

```

hello motan
110
11.2hello,motan,key3:[val1,val2,val3,val4,],key2:[val1,val2,val3,],key1:
[val1,val2,],
1001,motan bean,99.998

```

【代码解析】Motan 的服务方创建好服务接口之后，将暴露接口、端口等在配置文件中写好，启动配置文件即可开启，调用方需要使用与服务方相同的接口（包括包名相同），并正确配置配置文件，即可在代码中调用服务方暴露的接口服务。

### 3. 集群调用示例

实现集群调用只需要在上述操作的基础上做一点改变即可，Motan 的集群与阿里的 Dubbo 的原理类似，通过注册方、服务方、调用方三方来实现，三者的关系如图 6-12 所示。

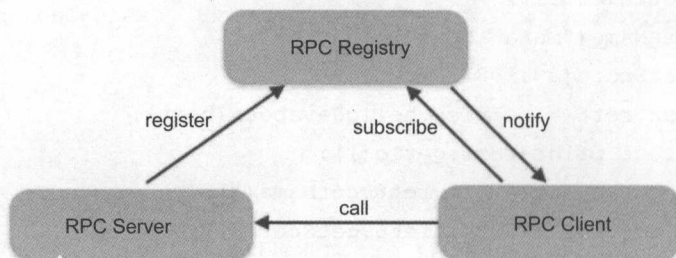


图 6-12 集群关系图

- RPC Server 向 RPC Registry 注册服务，并每隔一定时间向 RPC Registry 发送心跳汇报状态。
- RPC Client 需要向 RPC Registry 订阅 RPC 服务，RPC Client 根据 RPC Registry 返回的服务列表，对具体的 RPC Server 进行 RPC 调用。
- 当 RPC Server 发生变更时，RPC Registry 会同步变更，RPC Client 感知后会对本地的服务列表作相应调整。

目前按照 Wiki 说明，Motan 支持 Consul 和 Zookeeper 两种外部服务发现组件。

下面我们在上述实现的基础上进行更改（分别介绍两种组件）。

### （1）添加 pom 依赖。

前面已有介绍：

```
<!-- consul --><dependency>
  <groupId>com.weibo</groupId>
  <artifactId>motan-registry-consul</artifactId>
  <version>0.0.1</version></dependency><!-- zookeeper --><dependency>
  <groupId>com.weibo</groupId>
  <artifactId>motan-registry-zookeeper</artifactId>
  <version>0.0.1</version></dependency>
```

### （2）在 Server 和 Client 的配置文件中分别增加 registry 定义。

consul:

```
<motan:registry regProtocol="consul" name="my_consul" address="127.0.0.1:8500"/>
```

zookeeper 单节点:

```
<motan:registry regProtocol="zookeeper" name="my_zookeeper" address="127.0.0.1:2181"/>
```

zookeeper 多节点集群:

```
<motan:registry regProtocol="zookeeper" name="my_zookeeper" address="127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183"/>
```

### （3）将 motan client 及 server 配置改为通过 registry 服务发现。

consul client:

```
<motan:referer id="remoteService" interface="quickstart.FooService"
registry= "my_consul"/>
```

consul server:

```
<motan:service interface="quickstart.FooService" ref="serviceImpl" registry=
"my_consul" export="8002" />
```

zookeeper client:

```
<motan:referer id="remoteService" interface="quickstart.FooService"
registry= "my_zookeeper"/>
```

zookeeper server:

```
<motan:service interface="quickstart.FooService" ref="serviceImpl"
registry=" my_zookeeper" export="8002" />
```

(4) 调用方调用服务。

consul 需要显示调用心跳开关注册到 consul (zookeeper 不需要) :

```
MotanSwitcherUtil.setSwitcher(ConsulConstants.NAMING_PROCESS_HEARTBEAT_
SWITCHER, true)
```

## 6.6 总结

不同的游戏在服务器的逻辑需求是不相同的，本章只是给出一种服务器逻辑实现的思路，希望读者在实际应用中可以灵活使用。一个好的逻辑架构是一个游戏服务器成功的基本保证。

# 第7章

## 游戏安全

从第一次软件被黑客攻破开始，软件产业的安全问题就一直存在，谁都不敢保证一款游戏不会被破解。因为每当我们找到一个破解方式的防范方法时，就会产生新的破解方式。虽然做不到百分之百的破解防护，但我们也应该最大限度地保障游戏服务器的安全、稳定。

### 7.1

#### 游戏安全的必要性

对于人来说，安全就是人身不受到威胁，周围没有危险、自己也没有损失。总的来说，安全就是能够和谐相处，互不伤害，也没有造成危害的隐患。这种状态是免除了一切不能接受的损害风险的状态。安全，就是在生产过程中，把系统的运行状态对人的损害都控制在人能够接受的状态。

安全需求主要指人们对个人安全、健康保护、资源拥有、财产所有权、道德安全、工作安全、社会保障等方面的需求。

网络游戏作为一个虚拟社会，人们对其安全的需求有以下几点。



- (1) 游戏账号角色的安全需求。
- (2) 虚拟财产安全需求。
- (3) 公平竞争的需求。
- (4) 对健康良性的游戏公共环境的诉求。
- (5) 对游戏整体稳定运营的需求。

但是现在外挂猖獗，各种网络游戏中的账号被盗取、游戏中的财产被盗号者转移，以及各种外挂、刷游戏的工作室和游戏中出现的虚假恶意消息等安全问题，都严重影响了玩家对游戏的安全需求。

如果玩家得不到游戏安全的需求，会有很严重的后果。从以往的统计结果来看，游戏安全问题是导致玩家流失的主要因素，严重影响了游戏的健康发展。

游戏安全问题是游戏开发中不可忽视的问题。下面从技术的角度来讲解游戏中的安全防范。

## 7.2 登录安全

对于手游和页游来说，近几年的游戏几乎都是以联运方式运营的，游戏服务器本身不再保存用户密码，用户通过平台和游戏服务器的接口对接登录，接口做加密认证。虽然账号密码的安全问题可以暂时不管，但也要注意登录认证的 hash 字符串，如登录 hash 字符串的生效时间、hash 字符串的加密参数来源等，这样，恶意的登录也很难通过服务器的验证。

## 7.3 游戏充值

游戏充值一般也是众多平台的联运，需要与各个公司做接口对接，且接口规范各种各样，开发商必须遵循联运平台的接口规范。大致的充值流程如下。

- (1) 客户端向开发商服务器请求生成订单。
- (2) 客户端调用联运 SDK 打开支付界面，玩家输入支付信息。
- (3) 联运 SDK 将订单信息发送给联运服务器。
- (4) 联运服务器记录订单并发送订单给开发商服务器进行支付验证。
- (5) 开发商服务器校验订单信息，校验成功就对客户端发货，并返回校验结果给联运服务器。
- (6) 联运服务器返回订单结果给客户端。

以上为游戏充值的大概流程，作为开发商服务器，我们能做到的安全校验就是第 5 步，我们需要对订单信息进行校验，验证支付信息的合法性，验证通过才能视为订单成功。

一般联运商的接口中会提供一个游戏开发商的自定义参数，开发商可通过这个参数传递自定义的参数，我们可以在此基础上传递一些验证信息以加强安全策略，如随机参数、随机数字等。一般来说，只要在支付接口校验中做好验证工作，就可防范大部分支付接口的 bug 利用。

值得一提的是，iOS 的 App Store 支付已出现一种破解工具，可以直接使支付完成，并流传于大部分的越狱手机中。对于这种类型的 bug，只需要加上服务器校验即可防止破解。加上服务器的支付验证之后，即使 App Store 被破解，服务器的支付订单状态也是无法被修改的，这样就能保证支付的安全有效。

## 7.4

## SQL 注入

SQL 注入主要针对 MySQL 这样的关系型数据库，主要指通过在具有调用数据库操作接口功能的地方，输入自定义的 SQL 语句达到操作数据库的目的。最常见的方法就是在登录时进行 SQL 注入，如果后台的登录接口代码如下：

```
public boolean login(String name,String pwd){
    String sql = "select * from Account where name='"+name+"' and pwd='"+pwd+"'";
    ResultSet rs = DBUtil.query(sql);
```

```
if(rs!=null){  
    return true;  
}  
return true;  
}
```

上述代码初看似乎没有任何问题，如果传入的参数如下：

```
name:Henry  
pwd:123321
```

那么代入到 login 方法中，SQL 语句就变成了如下语句：

```
select * from Account where name='Henry' and pwd='123321';
```

当程序执行以上 SQL 语句时，如果这组用户名密码存在，则可以查询出登录是成功还是失败。

以上是没有进行 SQL 注入的正常操作，如果传入的参数如下：

```
name:Henry  
pwd:aaa' or '1'='1
```

那么代入到 login 方法中，SQL 语句就变成了如下语句：

```
select * from Account where name='Henry' and pwd='aaa or '1'='1';
```

相信有一定 SQL 语法基础的用户已经看出来了，无论 name 等于什么，pwd 等于什么，or '1'='1'就能使这条 SQL 语句查询出所有的用户密码，自然，登录验证也是成功的，并且无论输入什么用户名，都能登录成功。攻击者通过改变 SQL 语句甚至可以查询出所有用户的账号密码，如果 SQL 语句中出现对表的更新操作，那么就会造成无法想象的灾难，因此，在对数据库操作的时候，需要做好 SQL 注入的防范。

防范的方法有很多，比如对参数进行单引号的替换操作，或者利用 JDBC 的预处理语句进行参数注入，以防止攻击者进行 SQL 注入。

## 7.5

### 通信协议与消息格式

部分高级的懂游戏运作原理的攻击者，可能会通过网络抓包等模拟出具有相同通信

协议的数据包，没有做防范的服务器很难将正常客户端发送的数据与攻击者模拟的游戏数据区分开，攻击者可以利用这个 bug 对服务器做任何自己想做的事情，在游戏中刷刷金币、刷刷钻石再也不是难事。

对于这类攻击防护，客户端与服务器要做好双方信息的加密解密，并且服务器也要对所有请求数据进行严格的校验。

## 7.6

### 整型溢出

在 Java 中，int 型变量的存放空间是 4 个字节，它的范围是-2147483648~2147483647。在 Java 的有符号 int 型中存储时，数的最高位为符号描述位，4 字节共 32 位，除去第一位的符号描述位，剩下的 31 位每个位能表示两个数字，所以 4 字节有符号的整数的表示范围为（分正数和负数）：负数为  $2^{31}$  个，即-1~-2147483648，正数为  $2^{31}$  个，即 1 到 2147483647。当程序的逻辑中出现了 int 型最大值以上的计算时，就有可能出现整型溢出，比如某商品的价格超过了 int 型的最大值时，那么这个 int 数字的符号位就会变为 1 及负整数，再进行商品支付时，减去负数就等于加上正数，于是就会出现得到商品，但货币不减少反而大量增加的情况。假如这一 bug 被玩家发现，便可通过购买此商品无限刷货币。

此类 bug 可以通过使用 long 来代替 int 类型来防止，并且在游戏的数值运算时，应尽量检验参与计算的数值是否符合要求，比如，是否均为正数。

## 7.7

### 并发请求

当游戏服务器同时在线人数达到一定数量时，就有可能出现大量的瞬时并发请求，当这些请求同时请求一个共享资源时，要注意对这个资源的加锁保护。比如游戏中有多个联盟，每个联盟有一个声望值，这个声望的数值就属于共享资源。当多个联盟成员对这个声望值进行操作时，就要注意对声望值的操作保护。比如，玩家 1 通过游戏操作获得联盟声望值，程序要将这个声望值累加到原来的声望值上，就会先获取现在的声望值，然后进行累加，再存储到缓存中。此时，若玩家 2 再重复玩家 1 的操作，就能完成联盟



声望值的两次增加。可是如果在玩家 1 获取到声望值后，还没有进行累加操作，玩家 2 就开始获取声望值，则玩家 2 获取到的是玩家 1 累加前的声望值，若玩家 1 的操作与玩家 2 的操作并发进行，很可能最后的声望值结果只是他们其中一个人的操作结果，另一个人的操作结果会被覆盖。

要解决游戏服务器中的并发请求，就需要做好共享资源的加锁保护，当玩家 1 请求了联盟声望值后，就对联盟声望值这个共享资源进行加锁，玩家 2 请求联盟声望值的时候，先获取联盟声望值的锁，发现资源被锁住，于是线程等待。玩家 1 的累加操作完成后，释放联盟声望值的锁，玩家 1 获取到锁被释放，操作继续进行。以上的步骤能够保证玩家 1 和玩家 2 的操作在其他部分可以并发进行，但在共享资源操作时必须有序进行。

在游戏服务器开发中，尤其是使用 Java 这种多线程编程语言时，要注意多个线程并发进行时对共享资源的同步处理。

## 7.8

### 逻辑漏洞

游戏逻辑漏洞通常是忽略了开发过程中逻辑流程的严谨性，如之前在 DNF 中出现的一个刷背包的 bug，游戏中的道具“云幂袖珍罐”，可以打开两件相同的游戏装备，获得游戏币的概率很小，打开的装备也是不值钱的。但如果打开金币，则有 5000 万、8000 万或 1 亿金币，按正常的网上交易价格计算，5000 万金币可以卖 400 元。

据玩家称，装备需要用背包存放，但目前的背包格子最多只有 48 个，即只有 48 件装备的存储空间。漏洞是利用背包的有限空间存放 47 件装备（存储满是无法打开罐子的），只留下一个格子的空间，而在开“云幂袖珍罐”出装备时，会因缺少空间，导致打开装备失效，而罐子不消失。玩家继续打开罐子，直到出现金币，但金币并没有占据背包的空间，所以打开成功罐子就会消失，玩家就可以去第三方平台出售金币，将其兑换成现金。

逻辑上的漏洞通常很难发现，这就需要开发人员在开发过程中随时注意游戏的逻辑严谨性，最大限度地减少逻辑漏洞的出现。

## 7.9 日志系统

无论在何种服务器系统中，日志系统都是必不可少的。玩家在游戏中的所有操作，服务器都必须记录到日志中，当出现各种安全事件时，日志系统可以作为数据回滚、交易纠纷等的处理依据，它也是游戏服务器准确的数据监控的来源。

## 7.10 总结

在游戏开发中，必须做好游戏服务器的各种安全防护，把所有对服务器请求的数据都视为外挂来进行处理，严格校验筛选，过滤掉对服务器有危害的数据，保证游戏服务器安全稳定地运行。

# 高级篇

## 游戏服务器的设计及优化

我们已经基本了解了服务器开发中的一些模块组成,本章将从架构的角度带领大家研究游戏服务器,从服务器架构演变过程,到各类游戏服务器的架构模型分析,以及最后的优化。本章架构分析部分仅以个人分析为准,因为我并不是具体的游戏项目组成员,所以我并不知道具体游戏的具体架构模型,但根据个人经验及前辈分析可以分析出大概的模型,为大家设计优秀的服务器架构提供一些思路和方法。

# 第8章

## 服务器架构分析

### 8.1

#### 服务器架构的演变过程

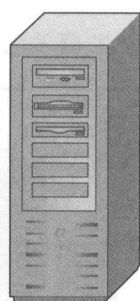
说到游戏服务器，就不得不说服务器的架构模型，由于网络游戏的需求日益变化，对服务器的要求也越来越高，最初的单服单节点早已不能满足如今网络游戏的需求。下面我们就来分析一下游戏服务器的架构演变过程。

最初的服务器并没有考虑那么多的需求，服务器也许就是最大在线人数几人或十几人的小型服务器，对人数、性能的要求都不高，这种服务器只需要把所有模块集中在一起即可，玩家的登录、逻辑操作、日志记录和数据库 IO 都集中部署在一台服务器上。

这种架构的服务器若不从硬件上进行改善，是不可能容下太多人的，而只通过改善硬件来实现服务器的拓展，成本又太高，并且一旦出现异常，服务器宕机，整个游戏数据几乎就是不可恢复的，这样的架构在现在并不可取（见图 8-1）。

为了保证游戏数据的完整，出现了逻辑与数据的分离，相比以前的服务器架构，这种架构更灵活，可根据服务器用途进行不同的配置，达到最佳性能效果。如果服务器宕机，那么至少能够保证数据仍然完整（见图 8-2）。





Game Server

图 8-1 单服务器架构图

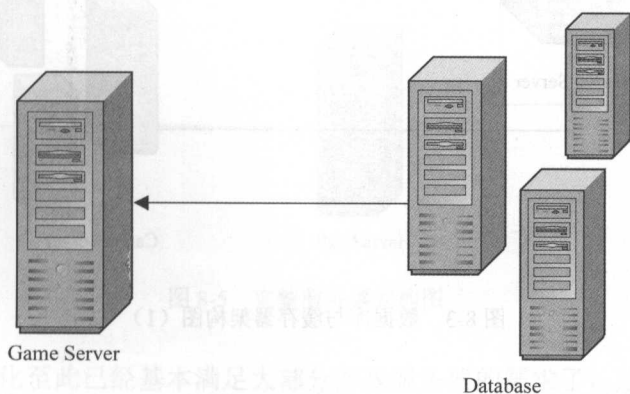


图 8-2 数据库集群服务器架构图

在通过硬件优化性能的同时，也可以通过软件进行性能优化，大部分游戏服务器都会利用缓存技术改善系统的性能。使用缓存主要源于热点数据的存在，大部分游戏数据都是临时热点数据，需要持久化的数据也许只占 20% 左右，所以可以对热点数据进行缓存，减少这些数据的访问路径，提高用户体验（见图 8-3）。

数据库集群服务器的结构已经做了很大的优化，可是如果在生产环境使用这种架构时你会发现，所有客户端请求，包括登录、支付、GM 管理、游戏逻辑等，全部由游戏服务器接入，这对单个服务器造成了巨大的压力。根据以上思路，我们可以继续细分架构，将游戏服务器分为管理服务器和逻辑服务器。逻辑服务器就是只负责处理游戏逻辑的服务器，管理服务器则负责处理游戏逻辑以外的登录、支付等其他的所有请求。这种架构能让管理服务器根据指定的分发策略对请求进行分流，将请求分发到不同逻辑服务器，实现负载均衡，也可在管理服务器实现选择服务器功能，根据用户的选择进入到不同的逻辑服务器（见图 8-4）。

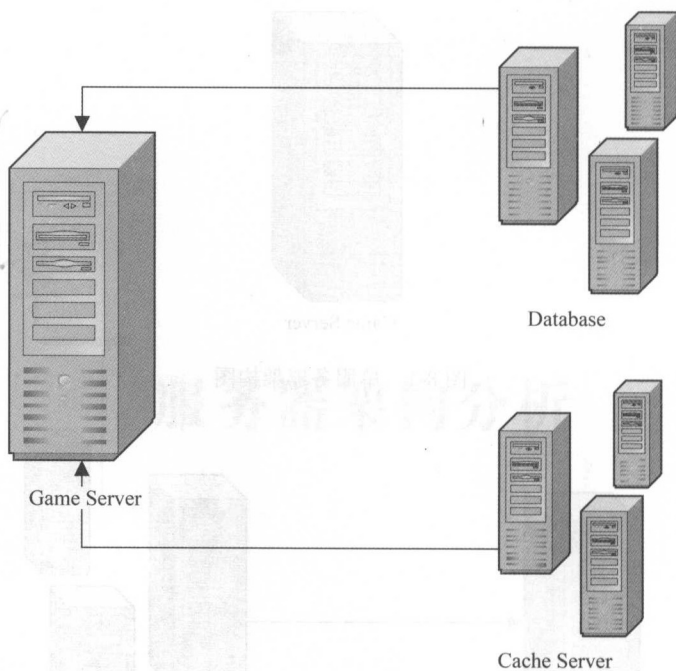


图 8-3 数据库与缓存器架构图 (1)

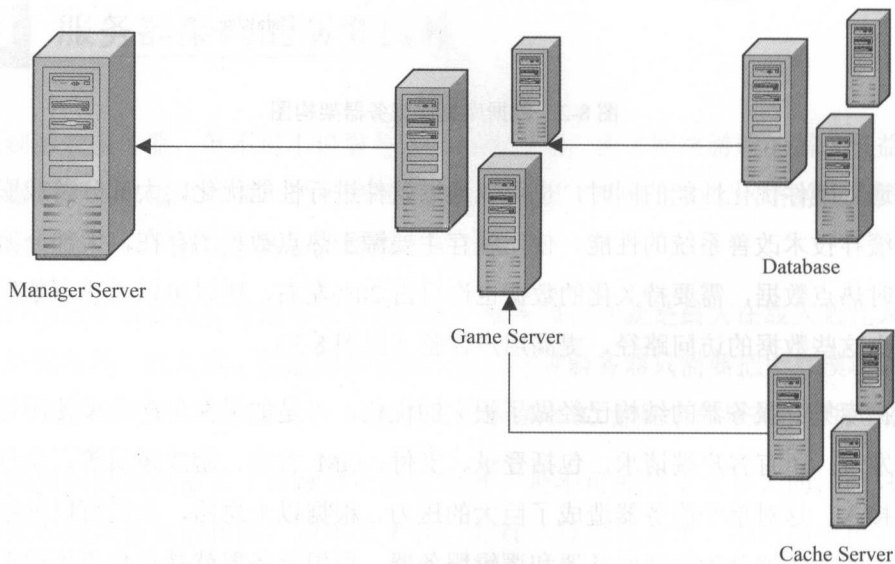


图 8-4 数据库与缓存器架构图 (2)

管理服务器处理了所有游戏逻辑以外的请求，使游戏体验好了很多，但所有压力都落在了管理服务器上，就会使登录速度变慢也会影响用户体验，而支付作为游戏唯一赚钱的途径，更要保证访问的速度，而且还有一些其他的请求，于是可以把管理服务器再

细分为支付服务器、登录服务器、GM 服务器（见图 8-5）。

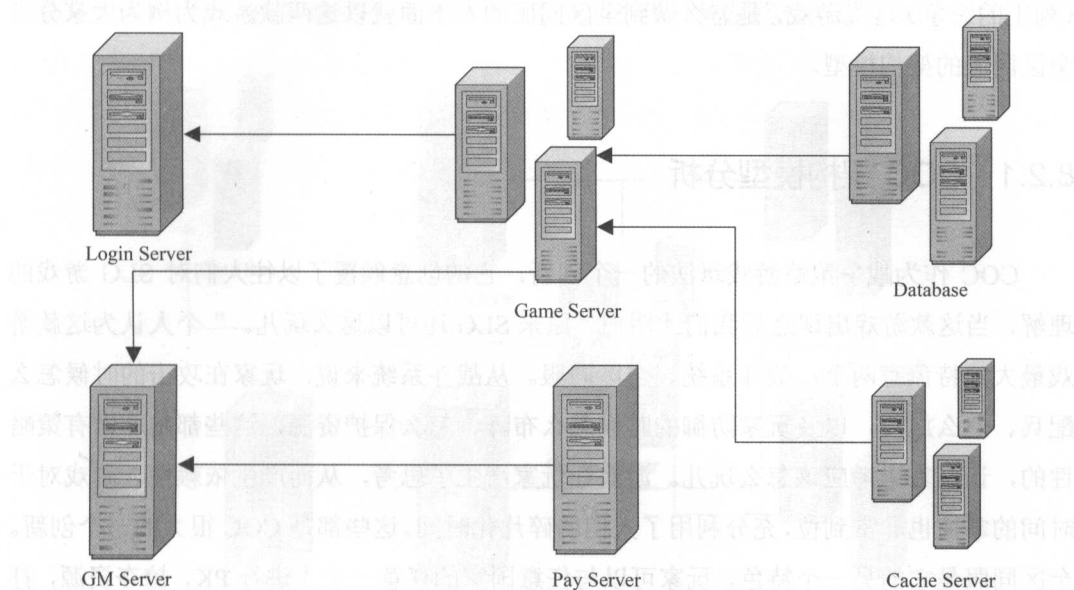


图 8-5 完整服务器架构图

服务器架构优化至此已经基本满足大部分游戏服务器的需求了，这样的架构对硬件的要求会降低许多，更多的是对服务器本身的性能要求，并且这种架构也很容易做到集群和扩展。如果数据库满足不了需求，可以对数据库再做主从分离；如果缓存不够用，可以再对缓存进行分布式处理；如果还需要 NoSQL 数据库，可以在 Database 服务器层面纵向扩展。这种架构也有很好的容灾能力，图 8-5 中任何一个节点宕掉，系统都能继续正常运行，比如某个游戏逻辑服务器出问题了，可以紧急关掉它，登录服务器会将登录请求分发到其他逻辑服务器；如果主数据库出现问题，数据库的操作请求也可以暂时分发到从数据库，紧急修复好问题节点，整个游戏系统就能正常运行。

只要处理好了游戏服务器架构的性能和容灾能力，玩家就会真正体验到游戏的稳定，从而提高玩家的依赖度和留存率。

## 8.2 全区同服架构分析

首先我们应该了解什么是全区同服游戏。网络游戏中，玩家经常希望和更多的人一起游戏，能不能不分服，让所有人一起打副本，一起 PK 呢？很遗憾的是，由于硬件水

平的限制,单个服务器的在线玩家承载数量是有一个上限的。那 COC(部落战争)、COK(列王的纷争)这类游戏,是怎么做到全区同服的?下面就以这两款游戏为例为大家分析全区同服的架构模型。

### 8.2.1 COC 架构模型分析

COC 作为战争策略游戏玩法的一个创新,它的创意颠覆了以往人们对 SLG 游戏的理解,当这款游戏出现之后我们才明白“原来 SLG 还可以这么玩儿。”个人认为这款游戏最大的特色有两个:战斗系统、全区同服。从战斗系统来说,玩家在攻击的时候怎么配兵、怎么放兵,以及玩家防御的时候怎么布阵、怎么保护资源,这些都是非常有策略性的,让玩家思考应该怎么玩儿。游戏让玩家产生了思考,从而产生依赖性。游戏对于时间的掌控也非常到位,充分利用了人们的碎片化时间。这些都是 COC 很大的一个创新。全区同服是它的另一个特色,玩家可以与任意国家的任意一个人进行 PK,掠夺资源,打部落战,这些都能充分调动玩家的兴趣。我们打开游戏时可以发现,它并没有让我们选服,也就是说,在我们看来,所有的玩家进的都是同一个服务器。全世界如此多的玩家,它又是怎么做到所有玩家都在同一个服务器的呢?实际上,只是我们看起来是同服而已,并不是所有玩家真的都在同一个服务器或同一个进程上。上文已经说过,单个服务器的玩家承载数量是有一个上限的,好的服务器能承载一万用户就已经很不错了。所谓的全区同服,实际上就是在架构上做好纵向扩展,让进程有更好的扩展性。玩家登录时,通过网关服务器连入网络延迟最低的逻辑服务器,而玩家的好友及全服排行榜则单独由一个全局服务器来处理,部落内聊天由一个聊天服务器单独处理,搜索匹配进行掠夺资源和部落战时由匹配服务器处理。架构分析模型如图 8-6 所示。

客户端登录由 Gate Server 接入,负责分发连接到最近的逻辑服务器;玩家的好友及排行榜等全局信息存储在 Global Server;部落内聊天的玩家消息广播由 Chat Server 完成;个人搜索匹配时,由于是直接打离线数据,直接通过 Search Server 在 Database 中寻找匹配的玩家,然后把数据拉过来,返回给逻辑服,打完之后返回结果并入库保存;而部落之间的战争,由于是临时的战斗队伍,作为热点数据,可以直接在 Cache Server 中存储正在匹配的部落,然后 Search Server 直接搜索 Cache Server 中也在匹配部落战的部落,两队的信息就可以暂时保存在 Search Server 中,战斗时两个部落所在的 Game Server(有可能两个部落在同一个逻辑服)和 Search Server 进行数据交互。游戏中看见的实时游戏



数据，实际上只是两个服务器之间阵形信息的传输，获取阵形信息之后，其他所有的画面都是由客户端在本地完成的战斗效果，最后战斗结果计算及奖惩信息将入库保存。

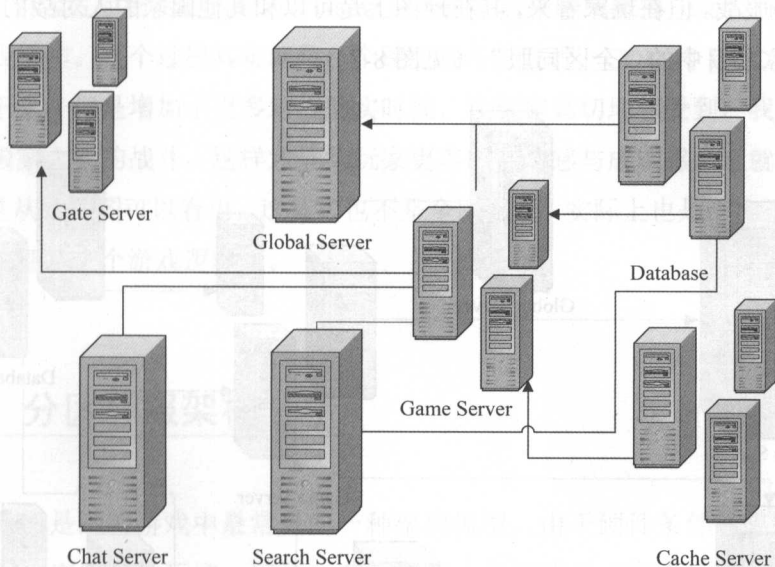


图 8-6 COC 服务器架构图

由以上架构我们可以知道，作为游戏核心的 Game Server 并非只有一个，所以所谓的全区同服只是服务器做了多层架构而已。

### 8.2.2 COK 架构模型分析

COK 将 SLG 带向了一个全新的方向。和 COC 相比，COK 的玩家交互性更强，不过这款游戏全区同服的感觉不像 COC 那么强烈，如国家之间的战争就属于跨服战。在世界地图上，玩家很容易发生冲突，假如两个领主发生了冲突，就会叫来各自的公会引发联盟战，联盟之间打得不相上下时，或许还会叫来友盟一起战斗，这样整个游戏的互动性就都调动起来了。能促成更多玩家之间交互的，便是全区同服。当我们第一次进入游戏时，会把我们分配到一个国家，5 级之前能迁移国家，5 级之后便只能在这个国家发展了。点开世界地图，能看到这个国家里所有的领主及其各自的联盟。地图中还有各种农田、伐木场、铁矿等资源田，领主可以派部队出去采集，但是一旦跨越这个国家的边境，界面就会开始出现加载 UI，然后才能看到这个国家的世界地图，并且其他国家的资源田，我们是不能采集的。从这种情况我们大概就能猜到，COK 中每一个国家便是一个服，查

看其他服的信息时，需要获取跨服的信息。游戏中时常会有很多活动，其中有一个活动叫做远古战场，进入这个游戏，就可以和世界地图中其他国家的人进行对抗，这其实就是 COK 的跨服战。但在玩家看来，其在地图上是可以和其他国家的人对战的，所以 COK 也就成了玩家心目中的“全区同服”（见图 8-7）。

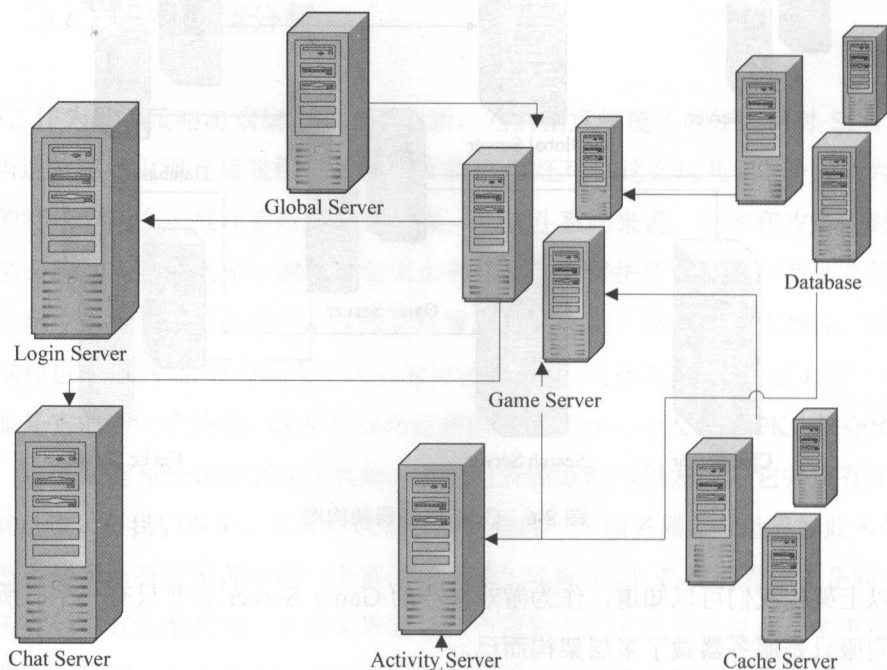


图 8-7 COK 服务器架构图

客户端登录由 Login Server 接入，如果是新用户，则会根据 COK 的分发策略（也许是分发到人数最少的国家，也许是分发到最新的国家）将用户分配到对应的国家（即对应的逻辑服）中，如果是老用户则直接进入相应的国家；游戏核心逻辑的相关操作在 Game Server 中完成；各种排行榜及一些全区信息，在 Global Server 中存储；国家内的聊天及联盟内的聊天由 Chat Server 来处理；游戏中阶段性的活动等相关内容，由 Activity Server 完成；国家内的战斗，均可以直接通过拉取 Database 或 Cache Server 中的玩家数据过来，进行数据计算并返回结果入库存储，客户端完成画面渲染即可。稍微复杂一点的就是跨服战，即远古战场，远古战场活动开启时会临时新增一个 Game Server（或者说这个 Game Server 一直存在，只是在远古战场开放时开服），各个国家（即各个 Game Server）参加远古战场的领主，都会被 Activity Server 将数据临时拉取到这个新增的 Game Server，然后领主之间的战斗都在这个临时的 Game Server 中完成，战斗结果再交给 Global Server

存储，最后根据国家战斗排名给各个国家的玩家发放奖励。具体的匹配规则也是在 Activity Server 中完成（跟 COC 的 Search Server 的作用一样，即根据制订的规则进行战斗匹配）。当远古战场活动结束后，Game Server 会被关服或撤掉，玩家回到自己的逻辑服继续游戏操作。整个过程其实就是一个跨服战斗的过程，COK 将 COC 打离线数据的模式保留下来，但是增加了更多对战的实时性，让玩家真切地感受到，我与盟友并肩参与了一场国家之间的战斗，这样才能给玩家更多的参与感与成就感，这就是全区同服的魅力。可是从架构图可以看出，这其实也不是全区同服，实际上也是分为了好多游戏服，即一个国家便是一个游戏逻辑服。

## 8.3 分区分服架构分析

分区分服是网络游戏中最常见的一种架构模型，由于硬件条件的限制，单服服务器只能承受一定数量的玩家，因此，多区多服才是游戏承载能力扩展的方式。一般情况下，分区分服的游戏，各个服务器之间没有交集，服务器之间的数据是独立的，互不影响。现在很多很火爆的网游都采用这种模式，大到英雄联盟、剑灵、CF，小到手游中的王者荣耀、全民超神，这些游戏的各个服务器之间没有任何联系，如果玩家不在一个区，就不可能一起玩游戏。这种游戏服务器架构的特点是，理论上可以纵向无限扩展逻辑服务器，来支撑日益增长的更多玩家。下面以大家最常玩的英雄联盟来分析举例。

英雄联盟自公测以来就备受喜爱，它在 DOTA 的 MOBA 模式上又做了一些创新，使游戏的操作更加简单，却又不失游戏的操作性，游戏的上帝视角、装备系统、匹配机制等，都是玩家感受非常棒的。进入英雄联盟之后，就会要求输入账号和密码，然后进行选服，这些操作就是在登录服务器完成的。进入游戏之后，就会进入到一个游戏大厅的界面，在这个游戏大厅，可以选择匹配模式和购买英雄等，这是在大厅服务器完成的，当你点击单人游戏或组队游戏之后，会匹配到玩家并进入游戏，这时便进入到核心的游戏逻辑服务器，并在逻辑服务器中完成所有游戏的相关操作。然而 MOBA 类网游最关键的操作还是对网络延迟的处理，怎样做到在物理上一定有延迟存在的情况下，还能让用户感受到即时操作的快感，是即时战斗游戏中最关键的部分。本部分只对服务器的架构模型进行分析，如图 8-8 所示。

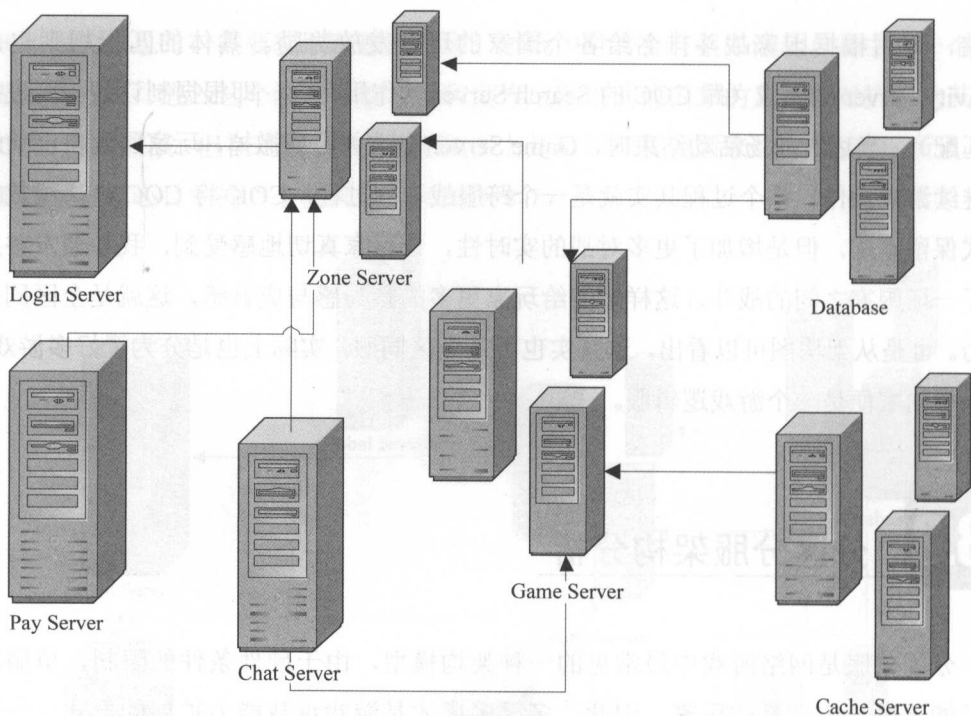


图 8-8 英雄联盟服务器架构图

客户端通过 Login Server 登录后，接入到大厅服务器 Zone Server。在大厅服务器中可以进行聊天和英雄皮肤购买，以及查看玩家战斗数据等，还负责玩家的匹配。Pay Server 负责游戏的商店信息及玩家的购买请求。Chat Server 负责聊天信息广播处理。当玩家匹配到一局游戏之后，便进入 Game Server 进行逻辑操作。一局游戏结束之后，便又回到最初的大厅服务器 Zone Server。这就是英雄联盟基本的服务器架构模型。

## 8.4 弱联网类游戏架构分析

近几年，手游市场异常火爆，不少手游公司如雨后春笋般出现。手游也由最初的单机手游发展为弱联网游戏，再发展为强联网游戏。提到弱联网游戏，大家一定会想到天天酷跑、天天飞车、节奏大师、全民消消乐等，这些游戏引领了一波手游热。那么这类游戏的服务器架构又是如何实现的呢？本节就以天天酷跑为例进行分析。

天天酷跑曾经是风靡一时的弱联网手游，在地铁上、公交车上，几乎处处都能看到拿着手机玩天天酷跑的人。这个游戏的创意原型实际上是 20 世纪八九十年代的超级玛



丽。那时的游戏产品虽然没有现在这种炫酷的画面，却仍风靡一时。在手游热火朝天的今天，将这款游戏翻做并加以创意，自然能受到不少玩家的喜爱。天天酷跑的服务器要实现的功能只有登录、支付、存档、回档等，大部分的逻辑运算及物品产出都是在客户端实现的，其服务器模型如图 8-9 所示。

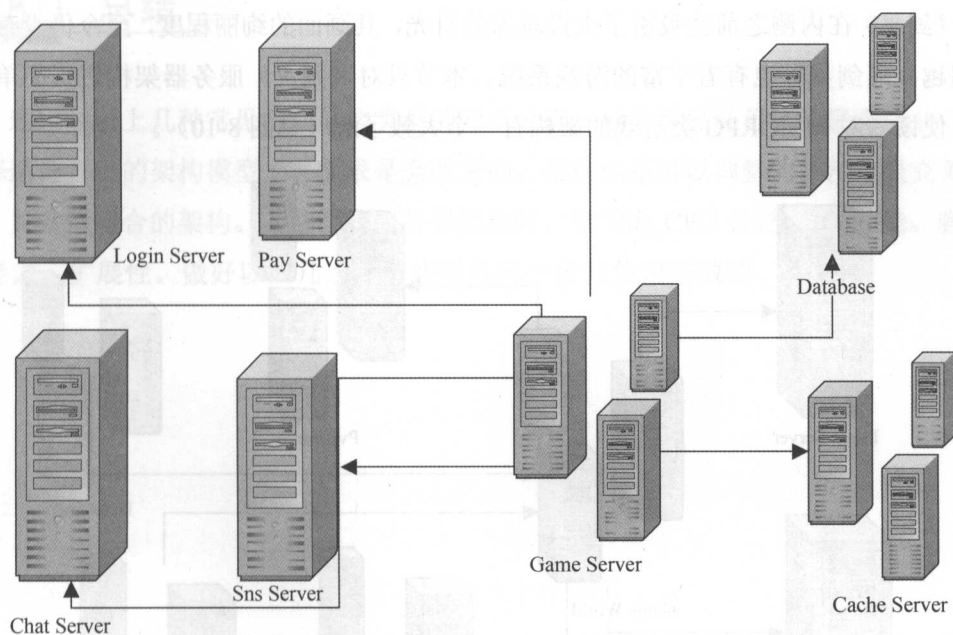


图 8-9 天天酷跑服务器架构图

服务器架构以 Game Server 为核心，客户端登录后由 Login Server 接入，然后进入 Game Server；游戏中的支付功能由 Pay Server 处理；社交系统，如好友、排行榜等由 Sns Server 来处理；Game Server 则在玩家进行游戏时存档，客户端需要读取服务器数据时由 Game Server 进行回档。整个服务器架构以 Game Server 为中心，呈星型散开，由于服务器需要做的处理很少，大部分操作都在客户端完成，因此这类游戏对服务器的要求并不高。

## 8.5

### MMORPG 类游戏架构分析

在弱联网手游出现之后，涌现了大量强联网手游，首当其冲的就是 MMORPG 类游戏。这类游戏对实时操作性要求较高，更注重玩家之间的交互感受。比如当初冲上 App Store 排行榜的天子、六龙争霸等，拼的几乎就是多少人能够同屏操作，所以我们经常看

到在游戏名后面会有一句“千人同屏”或“万人同屏”。要做到多少人同屏，不仅服务器要承受巨大的 IO 压力，客户端渲染也是一大瓶颈。实际上，MMORPG 的手游与端游的后台架构几乎是差不多的，其关键都在于网络 IO 的处理性能，以及服务器本身的处理性能。下面以《剑灵》为例来做架构分析。

《剑灵》在内测之前就吸引了大批玩家的目光，其画面的绚丽程度，至今仍没有网游能超越，《剑灵》也有着丰富的游戏系统。本节只对《剑灵》服务器架构进行简单的分析，使读者对 MMORPG 类游戏的架构有一个大致了解（见图 8-10）。

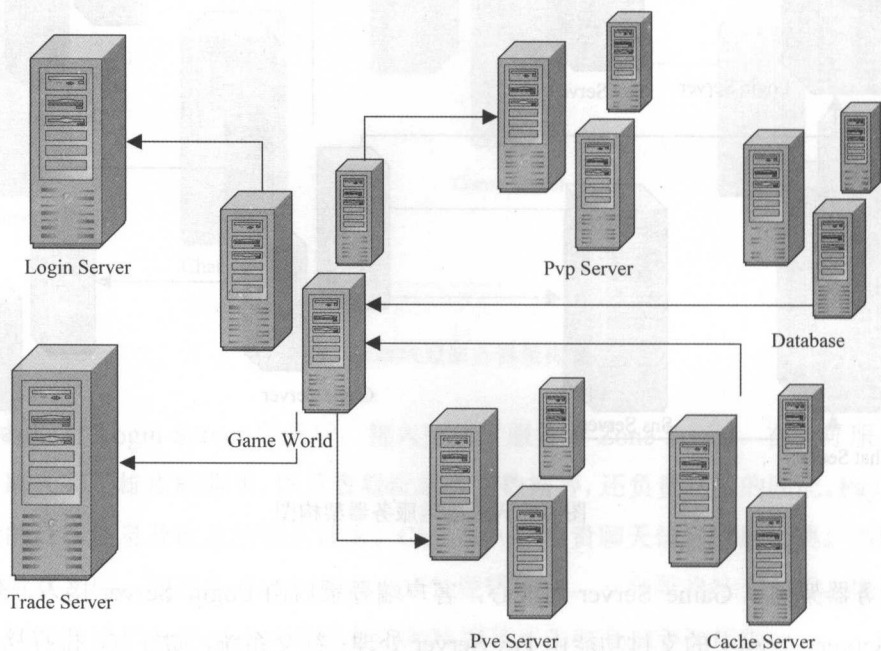


图 8-10 《剑灵》服务器架构图

图 8-10 只是《剑灵》后台架构的一个简单模型，还有很多系统模块并未展现出来。Login Server 负责客户端的接入及选服，Game World 是玩家的游戏世界，即地图中的世界，根据玩家所在位置，来同步地图中一定范围内此玩家的信息。当玩家选择 Pvp 战斗时，进入 Pvp Server，进行两个或多个玩家的 PK，玩家每进行一步操作，服务器都需要及时广播这个操作给所有玩家，其他玩家客户端即在本地图渲染出你做的这个操作，加上一些客户端预判等操作后，基本上能实现玩家之间零延迟的 PK。Pve 的原理与 Pvp 基本相同，最关键的部分就是玩家操作的及时广播，在对应情形下广播对应的信息，才能在最优性能的情况下同步玩家信息。除了核心的游戏世界及 Pvp、Pve 系统外，就是聊天系

统、交易系统、拍卖行等。MMORPG 最关键的部分，除了客户端本地渲染效果之外，就是服务器要做到玩家实时数据的交互。

## 8.6 总结

通过对以上几种常见游戏架构类型的简单分析，相信你已经可以根据实际业务需求选择适合自己的架构模型了。需求是会改变的，架构也是可以调整的。没有最完美的架构，只有最适合的架构。搭建游戏服务器架构时，要考虑 CPU 性能、IO 性能、容灾恢复能力、扩展性。做好以上几点，游戏服务器一定是稳定高效的。

## 第9章

# 《皇室战争》游戏开发实战

2016年1月，沉默了很久的游戏开发商 Supercell 在苹果 App Store 推出了一款微竞技手游——Clash Royal（以下简称《皇室战争》），同这家公司之前推出的 Clash Of Clans（以下简称《部落冲突》）一样，这款游戏一经推出就好评如潮，融合了卡牌、策略、养成、竞技等特色。同时也带领手游行业走入了一个新的模式——微竞技。本章以一个模仿《皇室战争》的简易 Demo 为例对 Java 游戏服务器开发进行介绍。

## 9.1 微竞技游戏介绍

《皇室战争》是如今较为流行的微竞技游戏之一，其开发商 Supercell 只是这款产品就有上亿的月流水，效益相当可观，可见微竞技游戏是能够充分满足当前玩家的需求的。《皇室战争》属于 RTS——即时战略游戏，它也融合了多种游戏元素，包括卡牌、即时战略、养成、竞技等特性，以增加微竞技的乐趣。



在端游的竞技游戏巅峰之时，微竞技手游也开始慢慢崛起。端游的传统竞技，如 CS、魔兽争霸、英雄联盟、DOTA 等，对玩家操作团队协作等都有较高的要求。微竞技的门槛相对较低，入手简单，但趣味性并未减少，通过游戏中简简单单的操作，就能获得巨大的竞技乐趣。

虽然人们现在越来越忙，但仍然有大量的碎片化时间无法利用，微竞技游戏的出现正好能填补人们无聊的碎片时间，正如《皇室战争》最流行的一句宣传语——没有什么事情是三分钟不能解决的，如果有，就再来一分钟。这句独具风格的宣传语包含了《皇室战争》最核心的内容——三分钟的战斗规则，每场竞技赛限时三分钟，如果三分钟未分出胜负，就再来一分钟的加时赛。这也是微竞技游戏的精髓，即在很短的时间内，完成一场竞技游戏，不需要进行复杂的操作，也不需要花费大量的时间。只要三分钟，你就能体会到竞技游戏的乐趣。

## 9.2

### 架构分析及搭建

从游戏服务器开发者的角度来看，《皇室战争》同《部落冲突》一样，是一种全球同服的游戏，拿起手机就可以和全世界任意一个国家、任意一个民族的人进行三分钟的战斗。在核心战斗部分，整个战斗过程是实时同步的，玩家所放的每一个英雄，每一个技能，在双方的屏幕中都能实时准确地出现，英雄之间进行战斗时的各种属性也是需要通过服务器进行实时同步的。

#### 9.2.1 功能分析

整个游戏以匹配系统和实时战斗系统为核心，也有卡牌养成系统和宝箱系统等模块。《皇室战争》的整个系统是非常庞大的，这里只写了一个简易版的系统，并且只有服务器端代码，客户端代码通过 Java Swing 来模拟。

通过对游戏功能模块的总结，可得出以下功能模块图（见图 9-1）。

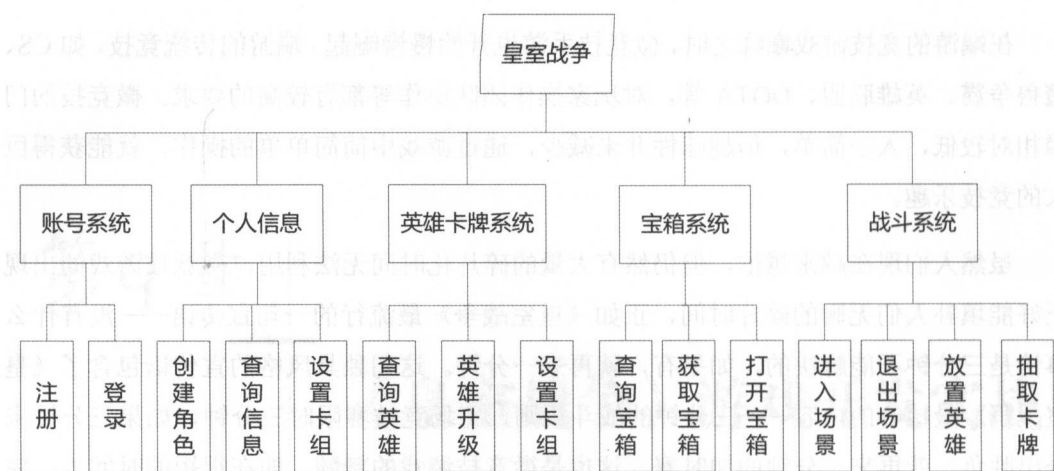


图 9-1 《皇室战争》功能模块图

## 9.2.2 服务器部署架构

我们可以将以上功能模块大致分为三个服务器系统：登录服务器（账号系统）、逻辑服务器（个人信息、英雄卡牌系统、宝箱系统）和对战服务器（战斗系统）。玩家登录时由登录服务器负责接入用户连接，然后给玩家分配逻辑服务器，如果要进入对战场景，再通过逻辑服务器获取到对战服务器的信息。三个服务器的部署架构关系如图 9-2 所示。

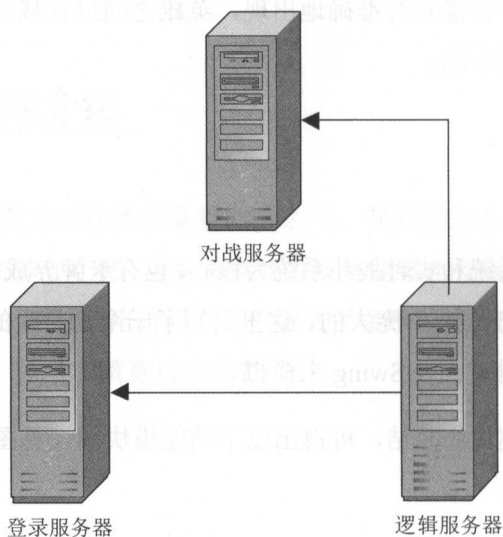


图 9-2 服务器部署架构图

### 9.2.3 系统架构

通过对《皇室战争》的分析，可以将整个游戏分为客户端、应用服务层与数据层三个层次。客户端负责前端逻辑处理及与服务器数据交互，应用服务层为游戏的逻辑处理服务，数据层对游戏的数据进行缓存或持久化。下面从几个方面对系统技术架构进行分析。

#### 1. 通信机制

根据《皇室战争》的特点分析，它在战斗部分属于 1V1 的实时战斗，需要使用 TCP 通信保证数据的实时性。而其他模块功能，如升级英雄、打开宝箱等，对实时性的要求相对较低，可以使用短连接 HTTP 通信，减少服务器的压力。HTTP 服务器与 TCP 服务器均采用 Netty5 实现。而登录服务器、逻辑服务器与对战服务器之间的通信，可以选择轻量的 Rpc 框架 Json-Rpc。将服务器按场景划分能降低场景之间的耦合度，具有更好的可扩展性与可维护性。

#### 2. 数据处理

玩家的数据是相对比较独立的部分，玩家与玩家之间的数据几乎没有关联性，并且玩家数据有良好的结构性，因此选择 MongoDB 的 BSON 数据格式存储玩家数据是一个比较好的选择。对于游戏中交互较频繁的热数据，需要使用 Redis 数据库进行存储，两者配合使用，能更好地处理服务器数据。Redis 的数据访问采用 Java API——Jedis，MongoDB 的数据访问采用 Morphia，Morphia 是一款类似 Hibernate 的 ORM 框架，它能使 MongoDB 的开发更容易。

#### 3. 项目管理

项目中用到了大量的第三方库，使用 Maven 管理项目的依赖包，以及项目的编译、发布、打包等，使项目管理更加容易，同时多个框架也可以使用 Spring 进行整合。

#### 4. 系统技术架构图

通过技术分析可总结出系统架构，如图 9-3 所示。

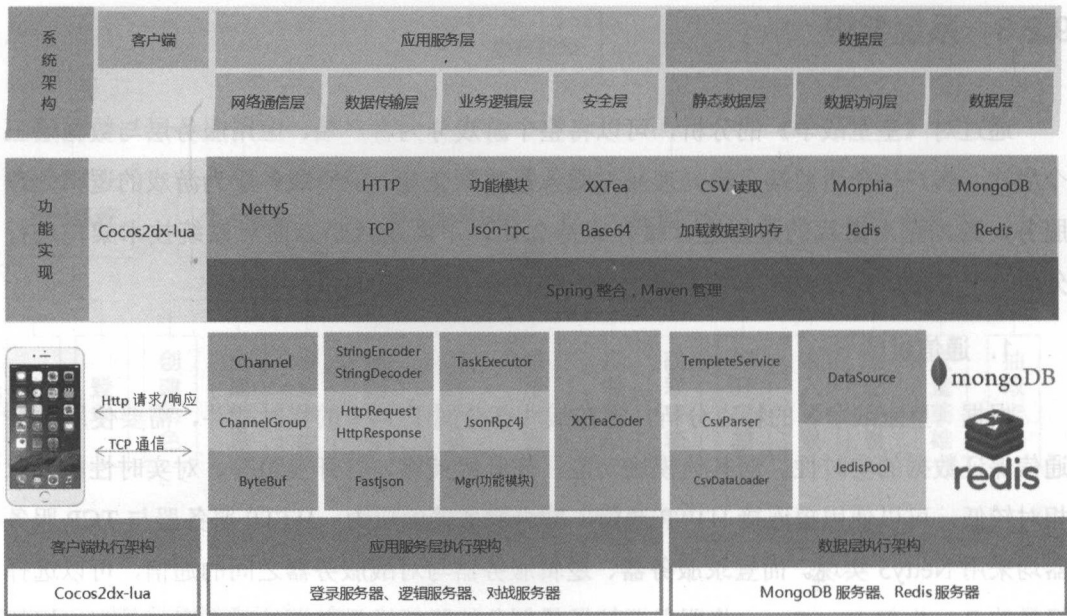


图 9-3 系统技术架构图

### 9.3 数据持久化方案

《皇室战争》的游戏数据相对来说是比较简单的，因为它的核心就是实时 PVP，更注重 PVP 的实时性，而不是个人养成模块。因此对于这款游戏，我们只要做好玩家个人数据的存储就可以了，PVP 中读取的数据都是进入实时对战场景就加载好的。

#### 9.3.1 数据结构分析

分析游戏的数据之后可以发现，多个玩家的数据之间没有关联性，而玩家自己的数据之间也具有很强的结构性，我们对玩家的数据进行分析之后得出如下结构（见图 9-4）。



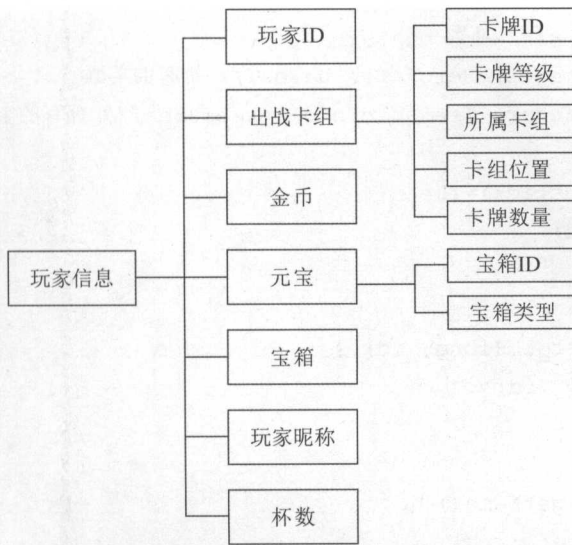


图 9-4 玩家数据结构图

我们可以将玩家数据结构映射为 JSON 对象，使用 MongoDB 的 BSON 结构可以更好地存储这类数据，并且具有很好的扩展性。因此建立基于以上数据结构的 JavaBean 如下。

Player:

```

package com.hjc.herol.manager.player;

import java.util.Map;

import org.mongodb.morphia.annotations.Entity;
import org.mongodb.morphia.annotations.Id;

import com.hjc.herol.manager.hero.HeroInfo;
import com.hjc.herol.manager.treasure.TreasureInfo;

@Entity
public class Player {
    @Id
    public long _id; // 用户 ID
    public String name; // 用户名
    public int coin; // 用户金币
    public int yuanbao; // 用户元宝
    public int fightGroup; // 出战卡组
  
```

```
public int cups;// 杯数
public Map<Integer, HeroInfo> heros;// 拥有的英雄
public Map<Integer, TreasureInfo> treasures;// 拥有的宝箱

public long get_id() {
    return _id;
}

public void set_id(long _id) {
    this._id = _id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getCoin() {
    return coin;
}

public void setCoin(int coin) {
    this.coin = coin;
}

public int getYuanbao() {
    return yuanbao;
}

public void setYuanbao(int yuanbao) {
    this.yuanbao = yuanbao;
}

public Map<Integer, HeroInfo> getHeros() {
    return heros;
}
```

```

public void setHeros(Map<Integer, HeroInfo> heros) {
    this.heros = heros;
}

public Map<Integer, TreasureInfo> getTreasures() {
    return treasures;
}

public void setTreasures(Map<Integer, TreasureInfo> treasures) {
    this.treasures = treasures;
}

public int getCups() {
    return cups;
}

public void setCups(int cups) {
    this.cups = cups;
}
}

```

#### HeroInfo:

```

package com.hjc.herol.manager.hero;

public class HeroInfo {
    private int heroId;
    private int level;// 卡牌等级
    private int fightGroup;// 出战卡组: 0-未出战, 1-1 卡组, 2-2 卡组, 3-3 卡组
    private int groupPosition;// 卡组位置
    private int count;

    public int getLevel() {
        return level;
    }

    public void setLevel(int level) {
        this.level = level;
    }
}

```

```
public int getFightGroup() {
    return fightGroup;
}

public void setFightGroup(int fightGroup) {
    this.fightGroup = fightGroup;
}

public int getHeroId() {
    return heroId;
}

public void setHeroId(int heroId) {
    this.heroId = heroId;
}

public int getGroupPosition() {
    return groupPosition;
}

public void setGroupPosition(int groupPosition) {
    this.groupPosition = groupPosition;
}

public int getCount() {
    return count;
}

public void setCount(int count) {
    this.count = count;
}
}
```

#### TreasureInfo:

```
package com.hjc.herol.manager.treasure;

public class TreasureInfo {
    private int id;
    private int type;
```



```

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public int getType() {
    return type;
}

public void setType(int type) {
    this.type = type;
}
}

```

【代码解析】以上几个 JavaBean 将上文分析的数据结构转换为了 Java 中的实体类，方便后面的数据操作。

### 9.3.2 使用 Morphia 操作 MongoDB

使用 MongoDB 时可以使用 Morphia 框架，它类似于关系型数据库的 Hibernate，对底层 API 进行了良好的封装，使我们在开发时能更专注业务逻辑，而不是数据库操作本身，Morphia 的封装类 MorphiaUtil 如下：

```

package com.hjc.herol.util.mongo;

import java.io.InputStream;
import java.net.UnknownHostException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;

import org.mongodb.morphia.Datastore;
import org.mongodb.morphia.Key;
import org.mongodb.morphia.Morphia;
import org.slf4j.Logger;

```

```
import org.slf4j.LoggerFactory;

import com.alibaba.fastjson.JSON;
import com.hjc.herol.manager.player.Player;
import com.mongodb.MongoClient;
import com.mongodb.MongoClientOptions;
import com.mongodb.ServerAddress;

public class MorphiaUtil {
    public static Logger logger = LoggerFactory.getLogger(MorphiaUtil.class);
    public static Datastore ds;
    private static final String CONF_PATH = "/spring-mongodb/mongodb.properties";
    public static String dbName = "db";

    public static Datastore getDatastore() {
        if (ds == null) {
            ds = buildDatastore();
        }
        return ds;
    }

    public static Datastore buildDatastore() {
        MongoClient mongo = null;
        try {
            String hosts = getProperty(CONF_PATH, dbName + ".host");
            mongo = new MongoClient(new ServerAddress(hosts.split(":")[0],
                Integer.parseInt(hosts.split(":")[1])),
                getDBOptions(dbName));
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
        Morphia morphia = new Morphia();
        morphia.mapPackage("com.hjc.herol");
        ds = morphia.createDatastore(mongo, "herol");
        ds.ensureIndexes();
        if (ds == null) {
            logger.error("mongo connect failed");
        }
    }
}
```

```

        return ds;
    }

    /**
     * @Title: getDBOptions
     * @Description: 获取数据参数设置
     * @return
     * @return MongoClientOptions
     * @throws
     */
    private static MongoClientOptions getDBOptions(String dbName) {
        MongoClientOptions.Builder build = new MongoClientOptions.Builder();
        build.connectionsPerHost(Integer.parseInt(getProperty(CONF_PATH,
dbName
        + ".connectionsPerHost"))); // 与目标数据库能够建立的最大
connection 数量为 50
        build.threadsAllowedToBlockForConnectionMultiplier(Integer
        .parseInt(getProperty(CONF_PATH, dbName
        +
        ".threadsAllowedToBlockForConnectionMultiplier"))); // 如果当前所有的
connection 都在使用中, 则每个 connection 可以有 50 个线程排队等待
        build.maxWaitTime(Integer.parseInt(getProperty(CONF_PATH, dbName
        + ".maxWaitTime")));
        build.connectTimeout(Integer.parseInt(getProperty(CONF_PATH, dbName
        + ".connectTimeout")));
        MongoClientOptions myOptions = build.build();
        return myOptions;
    }

    /**
     * @Title: main
     * @Description: Test Code
     * @param args
     *      void
     * @throws
     */
    public static void main(String[] args) {
        Datastore ds = MorphiaUtil.getDatastore();
        for (long i = 11; i <= 31; i++) {
            TestBean bean = new TestBean();
            bean.setId(i);

```



```

        bean.setMsg("test bean1");
        bean.setScore(100.2d);
        SubBean sub1 = new SubBean();
        sub1.setId(111);
        sub1.setStr("sub bean 1");
        SubBean sub2 = new SubBean();
        sub2.setId(121);
        sub2.setStr("sub bean 2");
        Map<Long, SubBean> subs = new HashMap<Long, SubBean>();
        subs.put(sub1.getId(), sub1);
        subs.put(sub2.getId(), sub2);
        bean.setSubBean(sub1);
        bean.setSubBeans(subs);
        Key<TestBean> key = ds.save(bean);
        logger.info(key.getCollection());
        logger.info(key.getClass() + "");
        logger.info(key.getId() + "");
        logger.info(key.getType() + "");
    }
    // query
    List<TestBean> list = ds.createQuery(TestBean.class).asList();
    for (TestBean testBean : list) {
        logger.info(JSON.toJSONString(testBean));
        // MorphiaUtil.deleteData(MorphiaUtil.ds.createQuery(TestBean.
class)
        // .field("_id").equal(testBean.getId())));
    }
    // update
    ds.update(
        ds.createQuery(TestBean.class).field("_id").equal(2),
        ds.createUpdateOperations(TestBean.class).set(
            "subBeans.11.str", "update map val"));
    // query
    List<TestBean> list2 = ds.createQuery(TestBean.class).asList();
    for (TestBean testBean : list2) {
        logger.info(JSON.toJSONString(testBean));
    }
}

/**

```



```

* 根据 properties 文件的 key 获取 value
*
* @param filePath
*       properties 文件路径
* @param key
*       属性 key
* @return 属性 value
*/
private static String getProperty(String filePath, String key) {
    Properties props = new Properties();
    try {
        InputStream in = MongoUtil.class.getResourceAsStream(filePath);
        props.load(in);
        String value = props.getProperty(key);
        return value;
    } catch (Exception e) {
        logger.info("load mongo properties exception {}", e);
        System.exit(0);
        return null;
    }
}

/**
* @param t
* @return
*/
public <T> Long getTableIDMax(Class<T> t) {
    long count = ds.find(Player.class).countAll();
    if (count <= 0) {
        count = 1;
    }
    return count;
}
}

```

MorphiaUtil 类中用到的数据库配置文件如下（开发阶段数据库中没有配置用户名和密码，可以忽略）：

```

#IP 和端口，多个主机用&相连
db.host=127.0.0.1:27017
#数据库名

```

```

db.database=war
#用户名
db.username=hjc
#密码
db.password=123321
#每个主机的最大连接数
db.connectionsPerHost=50
#线程允许最大等待连接数
db.threadsAllowedToBlockForConnectionMultiplier=50
#连接超时时间 1 分钟
db.connectTimeout=60000
#一个线程访问数据库时，在成功获取到一个可用数据库连接之前的最长等待时间为 2 分钟
#这里比较危险，如果超过 maxWaitTime 仍没有获取到连接，该线程就会抛出 Exception
#故这里设置的 maxWaitTime 应该足够大，以免由于排队线程过多造成数据库访问失败
db.maxWaitTime=120000

```

**【代码解析】**Morphia 的使用极大地简化了 Mongo 的数据库操作，通过 MorphiaUtil 的封装，操作 Mongo 可先获得 DataStore 对象，再通过 DataStore 对象调用 Morphia 的 API 进行操作。

## 9.4 Netty 网络框架的使用

无论是 HTTP 短连接通信，还是 TCP 长连接通信，都可以使用 Netty 实现底层的网络通信框架，只需要使用 Netty 相应的编解码处理器，即可正确处理相应协议的请求。

### 9.4.1 Netty 实现的 HTTP 服务器

在逻辑服务器中，所有的模块处理均通过 HTTP 实现，Netty 要实现 HTTP 服务器，只需要在添加处理器的时候添加 HttpRequestDecoder 与 HttpResponseEncoder，并添加响应自定义消息处理时使用的 HttpRequest 与 HttpResponse 进行请求和响应处理，其中 HttpServer 的代码如下：

```

package com.hjc.herol.net.http;

import io.netty.bootstrap.ServerBootstrap;

```

```
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.http.HttpObjectAggregator;
import io.netty.handler.codec.http.HttpRequestDecoder;
import io.netty.handler.codec.http.HttpResponseEncoder;
import io.netty.handler.stream.ChunkedWriteHandler;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.nio.charset.Charset;
import java.util.Properties;
import java.util.concurrent.Executors;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HttpServer {
    public static Logger log = LoggerFactory.getLogger(HttpServer.class);
    public static HttpServer inst;
    public static Properties p;
    public static String ip;
    public static int port;
    public static String pvpIp;
    public static int pvpPort;
    private NioEventLoopGroup bossGroup = null;
    private NioEventLoopGroup workGroup = null;

    private HttpServer() {

    }

    public static HttpServer getInstance() {
        if (inst == null) {
            inst = new HttpServer();
            inst.initData();
        }
    }
}
```



```

    }
    return inst;
}

public void initData() {
    try {
        p = readProperties();
        ip = p.getProperty("ip");
        port = Integer.parseInt(p.getProperty("port"));
        // PVP 服务器 IP 端口
        pvpIp = p.getProperty("pvpIp");
        pvpPort = Integer.parseInt(p.getProperty("pvpPort"));
    } catch (IOException e) {
        log.error("socket 配置文件读取错误");
        e.printStackTrace();
    }
}

public void start() {
    bossGroup = new NioEventLoopGroup(0, Executors.newCachedThreadPool()); // boss 线程组
    workGroup = new NioEventLoopGroup(0, Executors.newCachedThreadPool()); // work 线程组
    ServerBootstrap bootstrap = new ServerBootstrap();
    bootstrap.group(bossGroup, workGroup);
    bootstrap.channel(NioServerSocketChannel.class);
    bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        protected void initChannel(SocketChannel ch) throws Exception {
            ChannelPipeline pipeline = ch.pipeline();
            /* http request 解码 */
            pipeline.addLast("decoder", new HttpRequestDecoder());
            pipeline.addLast("aggregator", new HttpObjectAggregator(
(65536)));

            /* http response 编码 */
            pipeline.addLast("encoder", new HttpResponseEncoder());
            pipeline.addLast("http-chunked", new ChunkedWriteHandler());
            /* http response handler */
            pipeline.addLast("outbound", new HttpOutHandler());
            /* http request handler */

```



```

        pipeline.addLast("inbound", new HttpInHandler());
    }
});
log.info("端口{}已绑定", port);
bootstrap.bind(port);
}

public void shut() {
    if (bossGroup != null && workGroup != null) {
        bossGroup.shutdownGracefully();
        workGroup.shutdownGracefully();
    }
    log.info("端口{}已解绑", port);
}

/**
 * 读配置 socket 文件
 *
 * @return
 * @throws IOException
 */
protected Properties readProperties() throws IOException {
    Properties p = new Properties();
    InputStream in = HttpServer.class
        .getResourceAsStream("/net.properties");
    Reader r = new InputStreamReader(in, Charset.forName("UTF-8"));
    p.load(r);
    in.close();
    return p;
}
}

```

其中网络配置文件如下:

```

# local server
port = 8586
ip=127.0.0.1
# pvp server
pvpIp=127.0.0.1
pvpPort=8400
# task

```

```
handleTaskQueueCapacity=1000
handleTaskCorePoolSize=10
handleTaskMaxPoolSize=30
handleTaskKeepAliveSeconds=300
```

在自定义处理器中需要将请求解码,并交给核心路由类 Router 去处理游戏逻辑,将要发送的消息进行编码,再返回给请求的客户端,处理的代码如下:

```
package com.hjc.herol.net.http;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.http.DefaultFullHttpRequest;
import io.netty.handler.codec.http.DefaultFullHttpResponse;
import io.netty.handler.codec.http.FullHttpResponse;
import io.netty.handler.codec.http.HttpHeaders;
import io.netty.handler.codec.http.HttpMethod;
import io.netty.handler.codec.http.HttpResponseStatus;
import io.netty.handler.codec.http.HttpVersion;
import io.netty.handler.codec.http.QueryStringDecoder;
import io.netty.handler.codec.http.multipart.Attribute;
import io.netty.handler.codec.http.multipart.DefaultHttpDataFactory;
import io.netty.handler.codec.http.multipart.HttpPostRequestDecoder;
import io.netty.handler.codec.http.multipart.InterfaceHttpData;
import io.netty.util.CharsetUtil;

import java.io.UnsupportedEncodingException;
import java.net.URLDecoder;
import java.util.List;
import java.util.Map;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.hjc.herol.core.GameServer;
import com.hjc.herol.core.Router;
import com.hjc.herol.net.ProtoMessage;
```

```

import com.hjc.herol.net.ResultCode;
import com.hjc.herol.task.ExecutorPool;
import com.hjc.herol.util.Constants;
import com.hjc.herol.util.encrypt.XXTeaCoder;

public class HttpInHandlerImp {
    private static Logger log = LoggerFactory.getLogger(HttpInHandlerImp.class);
    public static String DATA = "data";
    public static volatile boolean CODE_DEBUG = false;

    public void channelRead(final ChannelHandlerContext ctx, final Object msg)
        throws Exception {
        /** work 线程的内容转交线程池管理类处理, 缩短 work 线程耗时 */
        ExecutorPool.channelHandleThreadPool.execute(new Runnable() {
            @Override
            public void run() {
                if (!GameServer.shutdown) { // 服务器开启的情况下
                    DefaultFullHttpRequest req = (DefaultFullHttpRequest) msg;
                    if (req.getMethod() == HttpMethod.GET) { // 处理 GET 请求
                        getHandle(ctx, req);
                    }
                    if (req.getMethod() == HttpMethod.POST) { // 处理 POST 请求
                        postHandle(ctx, req);
                    }
                } else { // 服务器已关闭
                    JSONObject jsonObject = new JSONObject();
                    jsonObject.put("errMsg", "server closed");
                    writeJSON(ctx, jsonObject);
                }
            }
        });
    }

    private void postHandle(final ChannelHandlerContext ctx,
        final DefaultFullHttpRequest req) {
        HttpPostRequestDecoder decoder = new HttpPostRequestDecoder(
            new DefaultHttpDataFactory(false), req);
        // 逻辑接口处理
        try {
            InterfaceHttpData data = decoder.getBodyHttpData(DATA);

```



```

        if (data != null) {
            String val = ((Attribute) data).getValue();
            val = codeFilter(val);
            log.info("ip:{},read:{}", ctx.channel().remoteAddress(), val);
            Router.getInstance().route(val, ctx);
        }
    } catch (Exception e) {
        log.error("post error msg:", e);
        e.printStackTrace();
        // Print our stack trace
        StringBuffer eBuffer = new StringBuffer(e.getMessage() + ",");
        StackTraceElement[] trace = e.getStackTrace();
        for (StackTraceElement traceElement : trace) {
            eBuffer.append("\r\n " + traceElement);
        }
        HttpInHandler.writeJSON(
            ctx,
            ProtoMessage.getErrorResp(ResultCode.SERVER_ERR,
                eBuffer.toString()));
    }
}

private void getHandle(final ChannelHandlerContext ctx,
    DefaultFullHttpRequest req) {
    QueryStringDecoder decoder = new QueryStringDecoder(req.getUri());
    Map<String, List<String>> params = decoder.parameters();
    List<String> typeList = params.get("type");
    if (Constants.MSG_LOG_DEBUG) {
        log.info("ip:{},read:{}", ctx.channel().remoteAddress(),
            typeList.get(0));
    }
    writeJSON(ctx, HttpResponseStatus.NOT_IMPLEMENTED, "not implement");
}

/**
 * @Title: codeFilter
 * @Description: 编解码过滤
 * @param val
 * @return
 * @throws UnsupportedEncodingException

```



```

*           String
* @throws
*/
private String codeFilter(String val) throws UnsupportedOperationException {
    val = val.contains("%") ? URLDecoder.decode(val, "UTF-8") : val;
    String valTmp = val;
    val = CODE_DEBUG ? XXTeaCoder.decryptBase64StringToString(val,
        XXTeaCoder.key) : val;
    if (Constants.MSG_LOG_DEBUG) {
        if (val == null) {
            val = valTmp;
        }
    }
    return val;
}

public static void writeJSON(ChannelHandlerContext ctx,
    HttpResponseStatus status, Object msg) {
    String sentMsg = null;
    if (msg instanceof String) {
        sentMsg = (String) msg;
    } else {
        sentMsg = JSON.toJSONString(msg);
    }
    sentMsg = CODE_DEBUG ? XXTeaCoder.encryptToBase64String(sentMsg,
        XXTeaCoder.key) : sentMsg;
    writeJSON(ctx, status,
        Unpooled.copiedBuffer(sentMsg, CharsetUtil.UTF_8));
    ctx.flush();
}

public static void writeJSON(ChannelHandlerContext ctx, Object msg) {
    String sentMsg = null;
    if (msg instanceof String) {
        sentMsg = (String) msg;
    } else {
        sentMsg = JSON.toJSONString(msg);
    }
    sentMsg = CODE_DEBUG ? XXTeaCoder.encryptToBase64String(sentMsg,
        XXTeaCoder.key) : sentMsg;

```

```

        writeJSON(ctx, HttpResponseStatus.OK,
            Unpooled.copiedBuffer(sentMsg, CharsetUtil.UTF_8));
        ctx.flush();
    }

    private static void writeJSON(ChannelHandlerContext ctx,
        HttpResponseStatus status, ByteBuf content /*
                                                    * , boolean
isKeepAlive
                                                    */) {

        if (ctx.channel().isWritable()) {
            FullHttpResponse msg = null;
            if (content != null) {
                msg = new DefaultFullHttpResponse(HttpVersion.HTTP_1_1, status,
                    content);
                msg.headers().set(HttpHeaders.Names.CONTENT_TYPE,
                    "application/json; charset=utf-8");
                msg.headers().set("userid", 101);
            } else {
                msg = new DefaultFullHttpResponse(HttpVersion.HTTP_1_1, status);
            }
            if (msg.content() != null) {
                msg.headers().set(HttpHeaders.Names.CONTENT_LENGTH,
                    msg.content().readableBytes());
            }
            // not keep-alive
            ctx.write(msg).addListener(ChannelFutureListener.CLOSE);
        }
    }

    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
        throws Exception {
        log.error("netty exception:", cause);
    }
}

```

**【代码解析】**使用 Netty 实现的 HTTP 服务器，通过 HttpRequestDecoder 和 HttpResponse Encode 对 HTTP 请求进行编解码，并通过 Netty 的 Channel 发送接收数据，操作完成则关闭通道，实现 HTTP 连接。

## 9.4.2 Netty 实现的 TCP 服务器

Netty 实现简单的 TCP 服务器只需要处理好消息的编解码, 以及 TCP 的各种参数设置即可, 这里我们使用 `StringEncoder` 与 `StringDecoder`, 代码如下:

```
package com.hjc.herolpvp.net.socket;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;
import io.netty.util.CharsetUtil;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.nio.charset.Charset;
import java.util.Properties;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SocketServer {
    public static Logger log = LoggerFactory.getLogger(SocketServer.class);
    public static SocketServer inst;
    public static Properties p;
    public static int port;
    private NioEventLoopGroup bossGroup = new NioEventLoopGroup();
    private NioEventLoopGroup workGroup = new NioEventLoopGroup();

    private SocketServer() {
    }
```



```
public static SocketServer getInstance() {
    if (inst == null) {
        inst = new SocketServer();
        inst.initData();
    }
    return inst;
}

public void initData() {
    try {
        p = readProperties();
        port = Integer.parseInt(p.getProperty("port"));
    } catch (IOException e) {
        log.error("socket 配置文件读取错误");
        e.printStackTrace();
    }
}

// Test Code
public static void main(String[] args) {
    SocketServer server = new SocketServer();
    SocketServer.port = 8700;
    server.start();
}

public void start() {
    ServerBootstrap bootstrap = new ServerBootstrap();
    bootstrap.group(bossGroup, workGroup);
    bootstrap.channel(NioServerSocketChannel.class);
    bootstrap.option(ChannelOption.SO_BACKLOG, 128);
    // 通过 NoDelay 禁用 Nagle, 使消息立即发出去, 不用等到一定的数据量才发出去
    bootstrap.option(ChannelOption.TCP_NODELAY, true);
    bootstrap.option(ChannelOption.SO_REUSEADDR, true);
    // 保持长连接状态
    bootstrap.childOption(ChannelOption.SO_KEEPALIVE, true);
    bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        protected void initChannel(SocketChannel ch) throws Exception {
            ChannelPipeline pipeline = ch.pipeline();
            pipeline.addLast(new StringDecoder(CharsetUtil.UTF_8));
        }
    });
}
```



```

        pipeline.addLast(new StringEncoder(CharsetUtil.UTF_8));
        // 业务逻辑处理
        pipeline.addLast(new SocketHandler());
    }
});
// 启动端口
ChannelFuture future;
try {
    future = bootstrap.bind(port).sync();
    if (future.isSuccess()) {
        log.info("端口{}已绑定", port);
    }
} catch (InterruptedException e) {
    log.info("端口{}绑定失败", port);
}
}

public void shut() {
    workGroup.shutdownGracefully();
    workGroup.shutdownGracefully();
    // 关闭所有 channel 连接
    log.info("关闭所有 channel 连接");
    ChannelMgr.getInstance().closeAllChannel();
    log.info("端口{}已解绑", port);
}

/**
 * 读配置 socket 文件
 *
 * @return
 * @throws IOException
 */
protected Properties readProperties() throws IOException {
    Properties p = new Properties();
    InputStream in = SocketServer.class
        .getResourceAsStream("/net.properties");
    Reader r = new InputStreamReader(in, Charset.forName("UTF-8"));
    p.load(r);
    in.close();
    return p;
}

```

```

    }
}

```

其中，网络配置文件如下：

```

# local server
port = 8400
ip=127.0.0.1
# task
handleTaskQueueCapacity=1000
handleTaskCorePoolSize=10
handleTaskMaxPoolSize=30
handleTaskKeepAliveSeconds=300

```

同样，在自定义处理器中也需要将请求解码，并交给核心路由类 Router 处理游戏逻辑，将要发送的消息进行编码，再返回给请求的客户端，处理代码如下：

```

package com.hjc.herolpvp.net.socket;

import java.io.UnsupportedEncodingException;
import java.net.URLDecoder;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelPromise;
import io.netty.channel.group.ChannelGroup;
import net.sf.json.JSONObject;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.alibaba.fastjson.JSON;
import com.hjc.herolpvp.core.GameServer;
import com.hjc.herolpvp.core.Router;
import com.hjc.herolpvp.manager.fight.FightMgr;
import com.hjc.herolpvp.net.ProtoIds;
import com.hjc.herolpvp.net.ProtoMessage;
import com.hjc.herolpvp.task.ExecutorPool;
import com.hjc.herolpvp.util.Constants;
import com.hjc.herolpvp.util.encrypt.XXTeaCoder;

public class SocketHandlerImp {

```

```

private static Logger log = LoggerFactory.getLogger(SocketHandlerImp.class);
public static volatile boolean ENCRYPT_DECRYPT = false;

public void channelRead(final ChannelHandlerContext ctx, final Object msg)
    throws Exception {
    ExecutorPool.channelHandleThreadPool.execute(new Runnable() {
        @Override
        public void run() {
            if (!GameServer.shutdown) { // 服务器开启的情况下
                dataHandle(ctx, msg);
            } else { // 服务器已关闭
                JSONObject jsonObject = new JSONObject();
                jsonObject.put("errMsg", "server closed");
                SocketHandler.writeJSON(ctx, jsonObject);
            }
        }
    });
}

/**
 * @Title: codeFilter
 * @Description: 编解码过滤
 * @param val
 * @return String
 * @throws
 */
private String codeFilter(String val) {
    try {
        val = val.contains("%") ? URLDecoder.decode(val, "UTF-8") : val;
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    String valTmp = val;
    val = ENCRYPT_DECRYPT ? XXTeaCoder.decryptBase64StringToString(val,
        XXTeaCoder.key) : val;
    if (Constants.MSG_LOG_DEBUG) {
        if (val == null) {
            val = valTmp;
        }
    }
}

```



```

        return val;
    }

    /**
     * @Title: dataHandle
     * @Description: 数据处理
     * @param ctx
     * @param msg
     *          void
     * @throws
     */
    private void dataHandle(final ChannelHandlerContext ctx, final Object
msg) {
        String body = (String) msg;
        body = codeFilter(body);
        ProtoMessage data = null;
        try {
            data = JSON.parseObject(body, ProtoMessage.class);
        } catch (Exception e) {
            log.error("格式错误, 需 json 格式数据");
            SocketHandler.writeJSON(ctx,
                ProtoMessage.getErrorResp("json 格式错误," + body));
            return;
        }
        if (Constants.MSG_LOG_DEBUG) {
            if (data.getTypeid() != ProtoIds.TEST) {
                log.info("read :" + body);
            }
        }
        Router.getInstance().route(data, ctx);
    }

    public void disconnect(ChannelHandlerContext ctx, ChannelPromise promise)
        throws Exception {
        log.info("ip:{}断开连接", ctx.channel().remoteAddress());
        Long userid = ChannelMgr.getInstance().findByChannel(ctx.channel()).
userid;
        if (userid != null) {
            FightMgr.getInstance().exitPkSceneMap(userid);
            FightMgr.getInstance().exitWaitingUsers(userid);
        }
    }

```



```

    }
}

public void channelActive(ChannelHandlerContext ctx) {
    log.info("ip:{}建立连接", ctx.channel().remoteAddress());
}

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
    log.error("ip:" + ctx.channel().remoteAddress() + "抛出异常", cause);
    Long userid = ChannelMgr.getInstance().findByChannel(ctx.channel());
    userid;
    if (userid != null) {
        FightMgr.getInstance().exitPkSceneMap(userid);
        FightMgr.getInstance().exitWaitingUsers(userid);
    }
}

/**
 * @Title: writeJSON
 * @Description: 发送 JSON 消息
 * @param ctx
 * @param msg
 * @return void
 * @throws
 */
public static void writeJSON(ChannelHandlerContext ctx, Object msg) {
    if (msg == null || msg instanceof String) {
        ctx.writeAndFlush(msg);
    } else {
        String sentMsg = JSON.toJSONString(msg);
        if (ctx.channel().isWritable()) {
            ctx.writeAndFlush(sentMsg);
            log.warn("channelId:{}", ctx.channel().id().asShortText());
        }
    }
}

/**
 * @Title: writeJSON
 * @Description: 群发 JSON 消息

```

```

    * @param group
    * @param msg
    *      void
    * @throws
    */
    public static void writeJSON(ChannelGroup group, Object msg) {
        if (msg == null || msg instanceof String) {
            group.writeAndFlush(msg);
        } else {
            String sentMsg = JSON.toJSONString(msg);
            group.writeAndFlush(sentMsg);
        }
    }

    public void write(ChannelHandlerContext ctx, Object msg) {
        if (Constants.MSG_LOG_DEBUG) {
            String resp = (String) msg;
            log.info("ip:{},write:{}", ctx.channel().remoteAddress(), resp);
        }
    }
}

```

【代码解析】通过 Netty 实现 TCP 服务器，实现游戏中实时数据的传输。

## 9.5 账号系统

每个网络游戏都有一套自己的账号系统，原版的《皇室战争》的账号系统即每个手机的 Game Center 账号，或者 Google Play 账号。进入游戏之后，直接由登录服务器给出分配逻辑服务器的信息，进入全球同服的服务器群。考虑到全球同服架构的复杂性，我们通过分服的方式进行逻辑服务器的选服登录。整个账号系统的注册、登录、选服功能均在登录服务器完成。我们可以通过 Spring MVC 实现登录服务器的功能，Spring 的 Controller 代码如下：

```

package com.hjc.herolrouter.client;

import java.io.IOException;
import java.util.Arrays;

```

```
import java.util.List;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONArray;
import com.alibaba.fastjson.JSONObject;
import com.hjc.herolrouter.client.model.Account;
import com.hjc.herolrouter.client.service.AccountService;
import com.hjc.herolrouter.server.NotifyService;
import com.hjc.herolrouter.server.ServerConfig;
import com.hjc.herolrouter.server.ServerService;
import com.hjc.herolrouter.util.Constants;
import com.hjc.herolrouter.util.hibernate.HibernateUtil;
import com.hjc.herolrouter.util.redis.Redis;

@Controller
@RequestMapping(value = "/route")
public class RouteController {
    @Autowired
    private AccountService accountService;
    @Autowired
    private ServerService serverService;
    @Autowired
    private NotifyService notifyService;
    public Logger logger = LoggerFactory.getLogger(RouteController.class);
    public static final String COUNTRY_CACHE = "country_cache";
    public static final String JUNZHU_CACHE = "junzhu_cache";

    @RequestMapping(value = "/loginOrRegist", method = RequestMethod.POST)
    public void loginOrRegist(HttpServletRequest request,
        HttpServletResponse response) {
```



```
String username = request.getParameter("name");
String password = request.getParameter("pwd");
JSONObject ret = new JSONObject();
if (username == null) {
    ret.put("code", 1);
    ret.put("msg", "用户名不能为空");
    writeJSON(ret, response, request.getRemoteAddr());
    return;
}
if (password == null) {
    ret.put("code", 1);
    ret.put("msg", "密码不能为空");
    writeJSON(ret, response, request.getRemoteAddr());
    return;
}
username = username.trim();
password = password.trim();
int state = accountService.loginOrRegist(username, password, 0);
ret.put("code", state);
switch (state) {
    case 100:
        ret.put("msg", "登录成功");
        break;
    case 101:
        ret.put("msg", "登录密码错误");
        break;
    case 200:
        ret.put("msg", "注册成功");
        break;
    case 201:
        ret.put("msg", "密码至少 6 位数");
        break;
}
if (state == 100 || state == 200) {
    Account account = accountService.getAccount(username);
    ret.put("userid", account.getId());
    ret.put("lastserver", account.getLastServer());
    List<ServerConfig> serverConfigs = HibernateUtil.list(
        ServerConfig.class, " order by id DESC");
    JSONArray servers = new JSONArray();
```



```

        for (ServerConfig serverConfig : serverConfigs) {
            JSONArray serverArr = new JSONArray();
            serverArr.add(serverConfig.getId());
            serverArr.add(serverConfig.getIp());
            serverArr.add(serverConfig.getPort());
            serverArr.add(serverConfig.getState());
            serverArr.add(serverConfig.getName());
            servers.add(serverArr);
        }
        ret.put("server", servers);
    }
    writeJSON(ret, response, request.getRemoteAddr());
}

@RequestMapping(value = "/regist", method = RequestMethod.POST)
public void regist(HttpServletRequest request, HttpServletResponse
response) {
    String username = request.getParameter("name");
    String password = request.getParameter("pwd");
    JSONArray ret = new JSONArray();
    if (username == null) {
        ret.add(1);
        ret.add("用户名不能为空");
        writeJSON(ret, response, request.getRemoteAddr());
        return;
    }
    if (password == null) {
        ret.add(1);
        ret.add("密码不能为空");
        writeJSON(ret, response, request.getRemoteAddr());
        return;
    }
    username = username.trim();
    password = password.trim();
    int state = accountService.regist(username, password, 0);
    switch (state) {
        case 0:
            ret.add(0);
            ret.add("注册成功");
            break;
    }
}

```

```
case 1:
    ret.add(1);
    ret.add("账号已存在");
    break;
case 2:
    ret.add(2);
    ret.add("密码至少 8 位数");
    break;
}
writeJSON(ret, response, request.getRemoteAddr());
return;
}

@RequestMapping(value = "/chooseServer", method = RequestMethod.POST)
public void chooseServer(HttpServletRequest request,
    HttpServletResponse response) {
    long userid = Long.parseLong(request.getParameter("userid"));
    long server = Long.parseLong(request.getParameter("server"));
    Account account = HibernateUtil.find(Account.class,
        (userid - server) / 1000);
    if (account != null) {
        account.setLastServer(server);
        HibernateUtil.save(account);
    }
    JSONObject result = new JSONObject();
    // 国家人数
    JSONArray arr = new JSONArray();

    // 国家人数
    String nums = Redis.getInstance().hget(Redis.GLOBAL_DB, COUNTRY_CACHE,
        String.valueOf(server));
    if (nums == null) {
        nums = "0#0#0";
    }
    String num1 = nums.split("#")[0];
    String num2 = nums.split("#")[1];
    String num3 = nums.split("#")[2];
    arr.add(num1);
    arr.add(num2);
    arr.add(num3);
}
```

```

// 玩家是否存在
String servers = Redis.getInstance().hget(Redis.GLOBAL_DB,
    JUNZHU_CACHE, String.valueOf(account.getId()));
servers = (servers == null) ? "-1#-1" : servers;
if (Arrays.asList(servers.split("#")).contains(String.valueOf(server))) {
    result.put("new", false);
} else {
    result.put("new", true);
    result.put("country", arr);
}
int sum = Integer.parseInt(num1) + Integer.parseInt(num2)
    + Integer.parseInt(num3);
result.put("sum", sum);
logger.info("账号{}登录了服务器{},服务器总人数{},魏国{}人,蜀国{}人,吴国{}人", userid, server,
    sum, Integer.parseInt(num1), Integer.parseInt(num2),
    Integer.parseInt(num3));
writeJSON(result, response, request.getRemoteAddr());
}

@RequestMapping(value = "/login", method = RequestMethod.POST)
public void login(HttpServletRequest request, HttpServletResponse
response) {
    String username = request.getParameter("name");
    String password = request.getParameter("pwd");
    JSONArray ret = new JSONArray();
    if (username == null) {
        ret.add(1);
        ret.add("用户名不能为空");
        writeJSON(ret, response, request.getRemoteAddr());
        return;
    }
    if (password == null) {
        ret.add(1);
        ret.add("密码不能为空");
        writeJSON(ret, response, request.getRemoteAddr());
        return;
    }
    username = username.trim();
    password = password.trim();

```



```
int state = accountService.login(username, password, 0);
switch (state) {
case 0:
    ret.add(0);
    ret.add("登录成功");
    Account account = accountService.getAccount(username);
    ret.add(account.getId());
    break;
case 1:
    ret.add(1);
    ret.add("账号不存在");
    break;
case 2:
    ret.add(2);
    ret.add("密码错误");
    break;
}
writeJSON(ret, response, request.getRemoteAddr());
}

@RequestMapping(value = "/getServers")
public void getServers(HttpServletRequest request,
    HttpServletResponse response) {
    List<ServerConfig> serverConfigs = HibernateUtil.list(
        ServerConfig.class, "");
    JSONArray ret = new JSONArray();
    for (ServerConfig serverConfig : serverConfigs) {
        JSONArray server = new JSONArray();
        server.add(serverConfig.getIp());
        server.add(serverConfig.getPort());
        server.add(serverConfig.getState());
        server.add(serverConfig.getName());
        ret.add(server);
    }
    writeJSON(ret, response, request.getRemoteAddr());
}

protected void writeJSON(Object msg, HttpServletResponse response,
    String remoteAddr) {
    String result = JSON.toJSONString(msg);
```



```

        if (Constants.CLIENT_DEBUG) {
            logger.info("ip:{},write:{}", remoteAddr, result);
        }
        try {
            response.getWriter().write(result);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

**【代码解析】**以上包括了注册登录和获取服务器、选取服务器的功能，玩家通过注册登录之后，调用获取服务器列表接口获取到所有的服务器信息，选择其中一个服务器进入，即转入到逻辑服务器的游戏场景。

通过登录服务器进入到逻辑服务器之后，就可以开始在游戏主界面操作调用各种逻辑接口，多种逻辑接口通过 `ProtoIds` 类进行管理，每个接口分配一个协议号，`Router` 根据不同协议号分配请求到不同逻辑模块，具体流程可参照前文讲到的游戏逻辑层的部分。

## 9.6 个人信息

个人信息即玩家的主要信息，包括玩家信息的查询、设置出战卡组 and 创建角色三个主要功能，相对应的协议号如下：

```

public static final short CREATE_ROLE = 1; // 创建角色
public static final short PLAYER_QUERY = 2; // 查询玩家信息
public static final short PLAYER_SET_GROUP = 3; // 设置出战卡组

```

在 `Router` 的 `switch` 路由分支如下：

```

/** 个人信息 **/
case ProtoIds.CREATE_ROLE:
    playerMgr.createRole(ctx, data);
    break;
case ProtoIds.PLAYER_QUERY:
    playerMgr.queryPlayer(ctx, data);

```

```
        break;
    case ProtoIds.PLAYER_SET_GROUP:
        playerMgr.setFightGroup(ctx, data);
        break;
```

其中, PlayerMgr 为个人信息的逻辑模块, 其完整的接口代码如下:

```
package com.hjc.herol.manager.player;

import java.util.HashMap;

import io.netty.channel.ChannelHandlerContext;

import org.mongodb.morphia.Datastore;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.alibaba.fastjson.JSONArray;
import com.hjc.herol.manager.hero.HeroInfo;
import com.hjc.herol.manager.hero.HeroMgr;
import com.hjc.herol.manager.treasure.TreasureInfo;
import com.hjc.herol.net.ProtoMessage;
import com.hjc.herol.net.http.HttpInHandler;
import com.hjc.herol.util.mongo.MorphiaUtil;

public class PlayerMgr {
    private static PlayerMgr junZhuMgr;
    private static final Logger logger = LoggerFactory
        .getLogger(PlayerMgr.class);
    private Datastore ds = MorphiaUtil.ds;

    private PlayerMgr() {
    }

    public static PlayerMgr getInstance() {
        if (null == junZhuMgr) {
            junZhuMgr = new PlayerMgr();
        }
        return junZhuMgr;
    }
}
```

```

public void initData() {
    logger.info("JunZhuMgr initData");
}

public void initCsvData() {
    logger.info("JunZhuMgr initCsvData");
}

/**
 * @Title: createRole
 * @Description: 创建角色
 * @param ctx
 * @param data
 * @param userid
 * @return void
 * @throws
 */
public void createRole(ChannelHandlerContext ctx, ProtoMessage data) {
    Player player = ds.find(Player.class).field("_id")
        .equal(data.getUserid()).get();
    if (player != null) {
        logger.error("角色已创建");
        HttpInHandler.writeJSON(ctx, ProtoMessage.getErrorResp("角色已
创建"));
        return;
    }
    player = new Player();
    player._id = data.getUserid();
    player.name = data.getData().getString("name");
    // 初始化金币 1000
    player.coin = 1000;
    // 初始化元宝 100
    player.yuanbao = 100;
    // 初始化 0 杯
    player.cups = 0;
    player.heros = new HashMap<Integer, HeroInfo>();
    player.treasures = new HashMap<Integer, TreasureInfo>();
    ds.save(player);
    // 默认添加三张英雄卡牌
    HeroMgr.getInstance().addHero(1, 1, data.getUserid());
}

```



```

HeroMgr.getInstance().addHero(2, 1, data.getUserid());
HeroMgr.getInstance().addHero(3, 1, data.getUserid());
HttpInHandler.writeJSON(ctx, ProtoMessage.getSuccessResp());
}

/**
 * @Title: queryPlayer
 * @Description: 查询玩家所有信息
 * @param ctx
 * @param data
 * @return void
 * @throws
 */
public void queryPlayer(ChannelHandlerContext ctx, ProtoMessage data) {
    Player player = ds.find(Player.class).field("_id")
        .equal(data.getUserid()).get();
    JSONArray ret = new JSONArray();
    ret.add(player._id);
    ret.add(player.name);
    ret.add(player.coin);
    ret.add(player.yuanbao);
    ret.add(player.fightGroup);
    ret.add(player.cups);
    // 添加英雄
    JSONArray heros = new JSONArray();
    for (Integer key : player.getHeros().keySet()) {
        JSONArray hero = new JSONArray();
        hero.add(player.getHeros().get(key).getHeroId());
        hero.add(player.getHeros().get(key).getLevel());
        hero.add(player.getHeros().get(key).getCount());
        hero.add(player.getHeros().get(key).getFightGroup());
        hero.add(player.getHeros().get(key).getGroupPosition());
        heros.add(hero);
    }
    ret.add(heros);
    // 添加宝箱
    JSONArray treasures = new JSONArray();
    for (Integer key : player.getTreasures().keySet()) {
        JSONArray treasure = new JSONArray();
        treasure.add(player.getTreasures().get(key).getType());
    }
}

```



```

        treasures.add(treasure);
    }
    ret.add(treasures);
    HttpInHandler.writeJSON(ctx, ret);
}

/**
 * @Title: setFightGroup
 * @Description: 设置出战卡组
 * @param ctx
 * @param data
 * @return void
 * @throws
 */
public void setFightGroup(ChannelHandlerContext ctx, ProtoMessage data) {
    long userid = data.getUserid();
    int fightGroup = data.getData().getIntValue("group");
    Player player = ds.find(Player.class).field("_id").equal(userid).
get();
    player.fightGroup = fightGroup;
    ds.save(player);
    HttpInHandler.writeJSON(ctx, ProtoMessage.getSuccessResp());
}
}

```

【代码解析】完成了个人模块创建角色、查询信息和设置卡组的功能。

## 9.7 英雄卡牌系统

英雄卡牌属于《皇室战争》比较重要的一个模块，是玩家战斗最主要的数据来源，包含三个主要功能：查询英雄、设置英雄所属卡组 and 升级英雄，相应的协议号如下：

```

public static final short HERO_SET_GROUP = 4; // 设置英雄所属卡组
public static final short HERO_QUERY = 5; // 查询英雄
public static final short HERO_UP_LEVEL = 6; // 升级英雄

```

在 Router 的 switch 分支如下:

```
/** 英雄卡牌 */
case ProtoIds.HERO_QUERY:
    heroMgr.queryHero(ctx, data);
    break;
case ProtoIds.HERO_SET_GROUP:
    heroMgr.setHeroGroup(ctx, data);
    break;
case ProtoIds.HERO_UP_LEVEL:
    heroMgr.upHeroLevel(ctx, data);
    break;
```

其中, HeroMgr 为英雄卡牌的逻辑模块, 其完整逻辑接口代码如下:

```
package com.hjc.herol.manager.hero;

import io.netty.channel.ChannelHandlerContext;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.mongodb.morphia.Datastore;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.alibaba.fastjson.JSONArray;
import com.hjc.herol.manager.player.Player;
import com.hjc.herol.net.ProtoMessage;
import com.hjc.herol.net.http.HttpInHandler;
import com.hjc.herol.template.Hero;
import com.hjc.herol.util.csv.TempletService;
import com.hjc.herol.util.mongo.MorphiaUtil;

public class HeroMgr {
    private static HeroMgr heroMgr;
    private static final Logger logger = LoggerFactory.getLogger(HeroMgr.class);
    private Datastore ds = MorphiaUtil.ds;
    public Map<Integer, Hero> heroMap = new HashMap<Integer, Hero>();
```

```

public static final int GROUP_MAX = 8; // 卡组最大卡牌数量

private HeroMgr() {
}

public static HeroMgr getInstance() {
    if (null == heroMgr) {
        heroMgr = new HeroMgr();
    }
    return heroMgr;
}

public void initData() {
    logger.info("HeroMgr initData");
}

public void initCsvData() {
    logger.info("HeroMgr initCsvData");
    // 加载英雄数据表，英雄数据读取自策划配置的 csv 数据表（这里是由我这个不大懂策划的人配置的，所以不必在意数据，主要是为了介绍数据加载的方式）
    Map<Integer, Hero> heroMap = new HashMap<Integer, Hero>();
    List<Hero> heros = TempletService.listAll(Hero.class.getSimpleName());
    for (Hero hero : heros) {
        heroMap.put(hero.getHeroId(), hero);
    }
    this.heroMap = heroMap;
}

/**
 * @Title: addHero
 * @Description: 添加英雄
 * @param heroId
 * @param count
 * @param userid
 * @return void
 * @throws
 */
public void addHero(int heroId, int count, long userid) {
    Player player = ds.find(Player.class).field("_id").equal(userid).
get();

```



```

        for (int i = 0; i < count; i++) {
            HeroInfo heroInfo = new HeroInfo();
            heroInfo.setLevel(1);
            heroInfo.setHeroId(heroId);
            heroInfo.setGroupPosition(0);
            heroInfo.setFightGroup(0);
            heroInfo.setCount(!player.getHeros().containsKey(
                Integer.valueOf(heroId)) ? 0 : player.getHeros()
                .get(heroId).getCount());
            player.getHeros().put(heroInfo.getHeroId(), heroInfo);
        }
        ds.save(player);
    }

    /**
     * @Title: setHeroGroup
     * @Description: 设置卡牌卡组
     * @param ctx
     * @param data
     * @return void
     * @throws
     */
    public void setHeroGroup(ChannelHandlerContext ctx, ProtoMessage data) {
        long userid = data.getUserid();
        int heroId = data.getData().getIntValue("heroid");
        int group = data.getData().getIntValue("group");
        int groupPos = data.getData().getIntValue("grouppos");
        Player player = ds.find(Player.class).field("_id").equal(userid).
        get();
        Map<Integer, HeroInfo> heros = player.getHeros();
        HeroInfo hero = heros.get(heroId);
        if (getHeroByGroup(heros, group).size() >= GROUP_MAX) {
            logger.error("玩家{}设置卡组失败, 卡组达到最大数量");
            HttpInHandler.writeJSON(ctx, ProtoMessage.getErrorResp("卡组达
            到最大数量"));
            return;
        }
        hero.setFightGroup(group);
        HeroInfo oldHero = getHeroByGroupPos(heros, groupPos);
        if (oldHero == null) {

```



```

        hero.setGroupPosition(groupPos);
    } else {
        // 位置上有卡牌, 替换到未出战卡组
        oldHero.setGroupPosition(0);
        oldHero.setFightGroup(0);
        heros.put(oldHero.getHeroId(), oldHero);
    }
    heros.put(heroId, hero);
    ds.save(player);
    HttpInHandler.writeJSON(ctx, ProtoMessage.getSuccessResp());
}

/**
 * @Title: getHeroByGroupPos
 * @Description: 根据卡组位置获取英雄
 * @param heros
 * @param groupPos
 * @return
 * @return HeroInfo
 * @throws
 */
public HeroInfo getHeroByGroupPos(Map<Integer, HeroInfo> heros, int
groupPos) {
    for (Integer key : heros.keySet()) {
        if (heros.get(key).getGroupPosition() == groupPos) {
            return heros.get(key);
        }
    }
    return null;
}

/**
 * @Title: upHeroLevel
 * @Description: 升级英雄卡
 * @param ctx
 * @param data
 * @return void
 * @throws
 */
public void upHeroLevel(ChannelHandlerContext ctx, ProtoMessage data) {

```

```

        long userid = data.getUserid();
        int heroId = data.getData().getIntValue("heroid");
        Player player = ds.find(Player.class).field("_id").equal(userid).
get();
        Map<Integer, HeroInfo> heros = player.getHeros();
        if (!heros.containsKey(Integer.valueOf(heroId))) {
            logger.error("玩家{}升级英雄失败, 没有英雄卡牌", userid);
            HttpInHandler.writeJSON(ctx, ProtoMessage.getErrorResp("没有英
雄卡牌"));
            return;
        }
        HeroInfo hero = heros.get(heroId);
        if (player.coin < 50 * hero.getLevel()) {
            logger.error("玩家{}升级英雄失败, 金币不足", userid);
            HttpInHandler.writeJSON(ctx, ProtoMessage.getErrorResp("金币不
足"));
            return;
        }
        if (hero.getCount() <= hero.getLevel() * 5) {
            logger.error("玩家{}升级英雄失败, 英雄数量不足", userid);
            HttpInHandler.writeJSON(ctx, ProtoMessage.getErrorResp("英雄数
量不足"));
            return;
        }
        hero.setLevel(hero.getLevel() + 1);
        hero.setCount(hero.getCount() - hero.getLevel() * 5);
        player.coin -= 50 * hero.getLevel();
        ds.save(player);
        HttpInHandler.writeJSON(ctx, ProtoMessage.getSuccessResp());
    }

    /**
     * @Title: queryHero
     * @Description: 查询英雄卡牌
     * @param ctx
     * @param data
     * @return void
     * @throws
     */
    public void queryHero(ChannelHandlerContext ctx, ProtoMessage data) {

```

```

        long userid = data.getUserid();
        int group = data.getData().getIntValue("group");// 要查询的出战卡组
        JSONArray ret = new JSONArray();
        Player player = ds.find(Player.class).field("_id").equal(userid).
get();
        Map<Integer, HeroInfo> heros = player.getHeros();
        ret.add(getHeroByGroup(heros, group));
        ret.add(getHeroByGroup(heros, 0));
        HttpInHandler.writeJSON(ctx, ret);
    }

    /**
     * @Title: getHeroByGroup
     * @Description: 获取卡组的卡牌
     * @param group
     * @param userid
     * @return
     * @return List<Integer>
     * @throws
     */
    public JSONArray getHeroByGroup(Map<Integer, HeroInfo> heros, int group) {
        JSONArray heroRet = new JSONArray();
        for (Integer key : heros.keySet()) {
            if (group == heros.get(key).getFightGroup()) {
                heroRet.add(heros.get(key).getHeroId());
                heroRet.add(heros.get(key).getLevel());
                heroRet.add(heros.get(key).getGroupPosition());
            }
        }
        return heroRet;
    }
}

```

**【代码解析】**完成了英雄模块中的查询英雄、设置英雄所属卡组 and 升级英雄功能。

在 `initCsvData` 方法中，读取了英雄卡牌的 CSV 数据配置表。在实际项目中，这些游戏中的静态数据应该交由数值策划进行配置，这里我直接做了一份简单的 CSV 英雄数据表，用于这个简单的 Demo。



## 9.8

## 宝箱系统

《皇室战争》中英雄卡牌的产出主要源自游戏中的宝箱系统，每次对战结束都能够随机获得一个宝箱，可能是木制宝箱、白银宝箱、黄金宝箱、超级宝箱或至尊宝箱，获得各个宝箱的概率是根据游戏中的表现决定的，在 Demo 中，我直接从 5 个宝箱中随机抽取。宝箱系统主要包含查询宝箱、打开宝箱和获取宝箱三个功能（其中，获取宝箱不需要暴露接口，只需要在战斗结束时调用即可）。

协议号定义如下：

```
public static final short HERO_UP_LEVEL = 6; // 升级英雄
public static final short TREASURE_QUERY = 7; // 查询宝箱
```

在 Router 的 switch 分支如下：

```
/** 宝箱系统 */
case ProtoIds.TREASURE_QUERY:
    treasureMgr.queryTreasure(ctx, data);
    break;
case ProtoIds.TREASURE_HERO_PICK:
    treasureMgr.pickHeroCard(ctx, data);
    break;
```

其中，TreasureMgr 是宝箱的逻辑模块，其完整接口代码如下：

```
package com.hjc.herol.manager.treasure;

import io.netty.channel.ChannelHandlerContext;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.mongodb.morphia.Datastore;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```



```

import com.alibaba.fastjson.JSONArray;
import com.hjc.herol.manager.hero.HeroMgr;
import com.hjc.herol.manager.player.Player;
import com.hjc.herol.net.ProtoMessage;
import com.hjc.herol.net.http.HttpInHandler;
import com.hjc.herol.template.Treasure;
import com.hjc.herol.util.csv.TemplateService;
import com.hjc.herol.util.mongo.MorphiaUtil;

public class TreasureMgr {
    private static TreasureMgr treasureMgr;
    private static final Logger logger = LoggerFactory
        .getLogger(TreasureMgr.class);
    private Datastore ds = MorphiaUtil.ds;
    public Map<Integer, Treasure> treasureMap = new HashMap<Integer,
Treasure>();
    public Map<Integer, Integer> treasurePropertyMap = new HashMap<Integer,
Integer>();
    public Map<Integer, Integer> propertyMap = new HashMap<Integer, Integer>();

    private TreasureMgr() {
    }

    public static TreasureMgr getInstance() {
        if (null == treasureMgr) {
            treasureMgr = new TreasureMgr();
        }
        return treasureMgr;
    }

    public void initData() {
        logger.info("TreasureMgr initData");
        // 三张卡的抽卡概率
        Map<Integer, Integer> treasurePropertyMap = new HashMap<Integer,
Integer>();
        treasurePropertyMap.put(1, 100);
        treasurePropertyMap.put(2, 40);
        treasurePropertyMap.put(3, 20);
        this.treasurePropertyMap = treasurePropertyMap;
        // 获取宝箱概率
    }
}

```

```

        Map<Integer, Integer> propertyMap = new HashMap<Integer, Integer>();
        propertyMap.put(1, 50);
        propertyMap.put(2, 20);
        propertyMap.put(3, 15);
        propertyMap.put(4, 10);
        propertyMap.put(5, 5);
        this.propertyMap = propertyMap;
    }

    public void initCsvData() {
        logger.info("TreasureMgr initCsvData");
        Map<Integer, Treasure> treasureMap = new HashMap<Integer,
Treasure>();
        List<Treasure> treasures = TempletService.listAll(Treasure.class
            .getSimpleName());
        for (Treasure treasure : treasures) {
            treasureMap.put(treasure.getType(), treasure);
        }
        this.treasureMap = treasureMap;
    }

    /**
     * @Title: pickHero
     * @Description: 抽取英雄卡
     * @param ctx
     * @param data
     * @return void
     * @throws
     */
    public void pickHeroCard(ChannelHandlerContext ctx, ProtoMessage data) {
        int treasureId = data.getData().getIntValue("treasureid");
        long userid = data.getUserid();
        JSONArray ret = new JSONArray();
        Player player = ds.find(Player.class).field("_id")
            .equal(data.getUserid()).get();
        Map<Integer, TreasureInfo> treasures = player.getTreasures();
        if (!treasures.containsKey(Integer.valueOf(treasureId))) {
            logger.error("玩家{}打开宝箱失败, 没有此宝箱", userid);
            HttpInHandler.writeJSON(ctx, ProtoMessage.getErrorResp("没有此
宝箱"));
        }
    }

```

```

        return;
    }
    Treasure treasure = treasureMap.get(treasureId);
    int random1 = (int) Math.round(Math.random() * 100);
    int random2 = (int) Math.round(Math.random() * 100);
    int random3 = (int) Math.round(Math.random() * 100);
    if (random1 < treasurePropertyMap.get(1)) {
        // 获取英雄 1
        int hero1 = treasure.getHero1();
        HeroMgr.getInstance().addHero(hero1, 1, userid);
        ret.add(hero1);
    }
    if (random2 < treasurePropertyMap.get(2)) {
        // 获取英雄 2
        int hero2 = treasure.getHero1();
        HeroMgr.getInstance().addHero(hero2, 1, userid);
        ret.add(hero2);
    }
    if (random3 < treasurePropertyMap.get(3)) {
        // 获取英雄 3
        int hero3 = treasure.getHero1();
        HeroMgr.getInstance().addHero(hero3, 1, userid);
        ret.add(hero3);
    }
    HttpInHandler.writeJSON(ctx, ret);
}

/**
 * @Title: pickTreasure
 * @Description: 获取宝箱
 * @param ctx
 * @param userid
 * @return void
 * @throws
 */
public void pickTreasure(long userid) {
    int random = (int) Math.round(Math.random() * 100);
    int sum = 0;
    int type = 0;

```



```

        for (int i = 1; i <= propertyMap.keySet().size(); i++) {
            int property = propertyMap.get(i);
            if (random >= sum && random < sum + property) {
                type = i;
                break;
            }
            sum += property;
        }
        Player player = ds.find(Player.class).field("_id").equal(userid).
get();
        TreasureInfo treasureInfo = new TreasureInfo();
        treasureInfo.setId(player.getTreasures().size() + 1);
        treasureInfo.setType(type);
        ds.save(player);
        logger.info("玩家{}抽取到宝箱{}", userid, type);
    }

    /**
     * @Title: queryTreasure
     * @Description: 查询宝箱
     * @param ctx
     * @param data
     * @return void
     * @throws
     */
    public void queryTreasure(ChannelHandlerContext ctx, ProtoMessage data) {
        long userid = data.getUserid();
        Player player = ds.find(Player.class).field("_id").equal(userid).
get();
        JSONArray ret = new JSONArray();
        Map<Integer, TreasureInfo> treasureMap = player.getTreasures();
        for (Integer key : treasureMap.keySet()) {
            ret.add(treasureMap.get(key).getType());
        }
        HttpInHandler.writeJSON(ctx, ret);
    }
}

```

**【代码解析】** 完成了宝箱系统，包括查询宝箱、打开宝箱和获取宝箱功能。



## 9.9 战斗系统

《皇室战争》最核心的部分是整个游戏的战斗系统,战斗系统需要转换到对战服务器。在转换之前,需要通过逻辑服务器获取到对战服务器的信息,对战服务器的信息写在逻辑服务器的配置文件中,启动服务器时读取,客户端可通过暴露的接口请求获取,协议号如下:

```
public static final short GET_PVP_SERVER = 10001; // 获取 PVP 服务器信息
```

在 Router 的 switch 分支如下:

```
case ProtoIds.GET_PVP_SERVER:
    fightMgr.getServer(ctx);
    break;
```

其中, FightMgr 提供对战服务器获取及战斗结算获取宝箱两个功能,其完整代码如下:

```
package com.hjc.herol.manager.fight;

import java.util.HashMap;
import java.util.Map;

import io.netty.channel.ChannelHandlerContext;

import org.mongodb.morphia.Datastore;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.alibaba.fastjson.JSONArray;
import com.hjc.herol.manager.player.Player;
import com.hjc.herol.manager.treasure.TreasureMgr;
import com.hjc.herol.net.ProtoMessage;
import com.hjc.herol.net.http.HttpInHandler;
import com.hjc.herol.net.http.HttpServer;
import com.hjc.herol.util.mongo.MorphiaUtil;
import com.hjc.herol.util.redis.Redis;
```

```
public class FightMgr {
    private static FightMgr fightMgr;
    private static final Logger logger = LoggerFactory
        .getLogger(FightMgr.class);
    public Redis redis = Redis.getInstance();
    public Map<Integer, Integer> cupMap = new HashMap<Integer, Integer>();
    private Datastore ds = MorphiaUtil.ds;

    private FightMgr() {
    }

    public static FightMgr getInstance() {
        if (null == fightMgr) {
            fightMgr = new FightMgr();
        }
        return fightMgr;
    }

    public void initCsvData() {
        logger.info("FightMgr initCsvData");
    }

    public void initData() {
        logger.info("FightMgr initData");
        // 战斗结算改变的杯数
        Map<Integer, Integer> cupMap = new HashMap<Integer, Integer>();
        cupMap.put(500, 60);
        cupMap.put(600, 50);
        cupMap.put(800, 40);
        cupMap.put(1000, 30);
        cupMap.put(1500, 20);
        cupMap.put(2000, 10);
        this.cupMap = cupMap;
    }

    /**
     * @Title: getServer
     * @Description: 获取 PVP 服务器
     * @param ctx
     * @return void
     */
}
```

```

    * @throws
    */
    public void getServer(ChannelHandlerContext ctx) {
        JSONArray ret = new JSONArray();
        ret.add(HttpServer.pvpIp);
        ret.add(HttpServer.pvpPort);
        HttpInHandler.writeJSON(ctx, ret);
    }

    /**
     * @Title: fightOver
     * @Description: 战斗结算
     * @param ctx
     * @param data
     * @return void
     * @throws
     */
    public void fightOver(ChannelHandlerContext ctx, ProtoMessage data) {
        long userid = data.getUserid();
        int result = data.getData().getIntValue("result");
        Player player = ds.find(Player.class).field("_id").equal(userid).
get();
        // 计算奖杯
        int sum = 0;
        int cup = 0;
        for (int i = 0; i < cupMap.keySet().size(); i++) {
            int cups = cupMap.get(cupMap.keySet().toArray()[i]);
            if (player.cups >= sum && player.cups < sum + cups) {
                cup = i;
                break;
            }
            sum += cups;
        }
        player.cups = result == 1 ? player.cups + cup : player.cups - (cup - 5);
        // 获取宝箱
        TreasureMgr.getInstance().pickTreasure(userid);
        ds.save(player);
    }
}

```



获取到对战服务器之后，需要与对战服务器建立 TCP 连接。对战服务器中进行游戏实时对战的全部处理，主要包括进入对战场景、退出对战场景、放置英雄、抽取英雄卡牌和英雄释放技能等功能。玩家进入场景之后，如果等待队列有人会直接取出等待队列中的人进行对战，如果等待队列没有人，就会进入等待队列等待玩家加入。游戏中的放置英雄和英雄释放技能需要对场景中的两个人进行同步。

协议号如下：

```
/** 登录请求 **/
public static final short TEST = 10000; // 测试网络延迟
public static final short EXIT_SCENE = 10002; // 退出场景
public static final short FIGHT_ENTER_SCENE = 1; // 进入游戏场景
public static final short FIGHT_SKILL = 2; // 释放技能
public static final short FIGHT_PICK_HERO = 3; // 放置英雄
```

对战服务器的 Router 如下：

```
/**
 * @Title: route
 * @Description: 消息路由分发
 * @param val
 * @param ctx
 *          void
 * @throws
 */
public void route(ProtoMessage data, ChannelHandlerContext ctx) {
    switch (data.getTypeId()) {
        case ProtoIds.TEST:
            test(ctx);
            break;
        case ProtoIds.EXIT_SCENE:
            fightMgr.exitScene(data.getUserid());
            break;
        case ProtoIds.FIGHT_ENTER_SCENE:
            fightMgr.enterScene(ctx, data);
            break;
        case ProtoIds.FIGHT_SKILL:
            fightMgr.skill(ctx, data);
            break;
        case ProtoIds.FIGHT_PICK_HERO:
```



```

        fightMgr.pickHero(ctx, data);
        break;
    default:
        logger.error("未知协议号:{}", data.getTypeid());
        SocketHandler.writeJSON(ctx,
            ProtoMessage.getErrorResp("未知协议号" + data.getTypeid()));
        break;
    }
}

```

FightMgr 的完整代码如下:

```

/**
 * @Title: FightMgr.java
 * @Package com.hjc.herolpvp.manager.fight
 * @Description: 实时战斗
 * @author 何金成
 * @date 2016年5月12日 下午12:14:36
 */
package com.hjc.herolpvp.manager.fight;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.group.ChannelGroup;
import io.netty.channel.group.DefaultChannelGroup;
import io.netty.util.concurrent.GlobalEventExecutor;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import java.util.Vector;
import java.util.concurrent.ConcurrentHashMap;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.alibaba.fastjson.JSONArray;
import com.googlecode.jsonrpc4j.JsonRpcHttpClient;
import com.hjc.herolpvp.core.GameInit;
import com.hjc.herolpvp.net.ProtoMessage;
import com.hjc.herolpvp.net.ResultCode;

```

```
import com.hjc.herolpvp.net.socket.ChannelMgr;
import com.hjc.herolpvp.net.socket.ChannelUser;
import com.hjc.herolpvp.net.socket.SocketHandler;

/**
 * @ClassName: FightMgr
 * @Description: TODO
 * @author 何金成
 * @date 2016年5月12日 下午12:14:36
 *
 */
public class FightMgr {
    private static FightMgr fightMgr;

    public Logger logger = LoggerFactory.getLogger(FightMgr.class);
    public ConcurrentHashMap<Long, ChannelGroup> pkSceneMap = new
ConcurrentHashMap<Long, ChannelGroup>();
    public List<Long> waitingUsers = new Vector<Long>();

    private FightMgr() {
    }

    public static FightMgr getInstance() {
        if (fightMgr == null) {
            fightMgr = new FightMgr();
        }
        return fightMgr;
    }

    /**
     * @Title: enterScene
     * @Description: 进入游戏场景
     * @param ctx
     * @param data
     * @return void
     * @throws
     */
    public void enterScene(ChannelHandlerContext ctx, ProtoMessage data) {
        long userid = data.getUserid();
        // 添加到 channel 管理
    }
}
```

```

ChannelMgr.getInstance().addChannelUser(ctx.channel(), userid);
// 匹配等待队列
if (waitingUsers.size() == 0) {
    if (pkSceneMap.containsKey(Long.valueOf(userid))) {
        SocketHandler.writeJSON(ctx,
            ProtoMessage.getErrorResp("玩家" + userid + "已经在
对战场景中"));
        return;
    }
    synchronized (waitingUsers) {
        // 没有人, 直接加入等待队列
        waitingUsers.add(userid);
    }
    SocketHandler.writeJSON(ctx, ProtoMessage.getSuccessResp());
    return;
}
if (waitingUsers.containsKey(Long.valueOf(userid))) {
    SocketHandler.writeJSON(ctx,
        ProtoMessage.getErrorResp("玩家" + userid + "已经在等待
队列中"));
    return;
}
if (pkSceneMap.containsKey(Long.valueOf(userid))) {
    SocketHandler.writeJSON(ctx, "玩家" + userid + "已经在对战场景中");
    return;
}
long fightUser = 0;
// 取出第一个人
synchronized (pkSceneMap) {
    fightUser = waitingUsers.remove(0);
    ChannelGroup pkScene = new DefaultChannelGroup(
        GlobalEventExecutor.INSTANCE);
    waitingUsers.remove(fightUser);
    pkScene.add(ctx.channel());
    pkScene.add(ChannelMgr.getInstance().getChannel(fightUser));
    pkSceneMap.put(userid, pkScene);
    pkSceneMap.put(fightUser, pkScene);
}
SocketHandler.writeJSON(pkSceneMap.get(Long.valueOf(userid)),
    ProtoMessage.getResp("玩家" + userid + "进入了对战场景",

```



```

        ResultCode.ENTER_PVP));
    SocketHandler.writeJSON(pkSceneMap.get(Long.valueOf(fightUser)),
        ProtoMessage.getResp("玩家" + fightUser + "进入了对战场景",
            ResultCode.ENTER_PVP));
}

/**
 * @Title: skill
 * @Description: 英雄释放技能
 * @param ctx
 * @param data
 * @return void
 * @throws
 */
public void skill(ChannelHandlerContext ctx, ProtoMessage data) {
    long userid = data.getUserid();
    int hero = data.getData().getIntValue("hero");
    SocketHandler.writeJSON(pkSceneMap.get(Long.valueOf(userid)), "
玩家"
        + userid + "的英雄" + hero + "释放了技能");
}

/**
 * @Title: attack
 * @Description: 放置英雄
 * @param ctx
 * @param data
 * @return void
 * @throws
 */
public void attack(ChannelHandlerContext ctx, ProtoMessage data) {
    long userid = data.getUserid();
    int hero = data.getData().getIntValue("hero");
    double x = data.getData().getDouble("x");
    double y = data.getData().getDouble("y");
    // TODO 对英雄所放置的位置做逻辑判断
    //
    SocketHandler.writeJSON(pkSceneMap.get(Long.valueOf(userid)), "
玩家"
        + userid + "把英雄" + hero + "放在了(" + x + "," + y + ")位置");
}

```



```

}

/**
 * @Title: pickHero
 * @Description: 从卡组中抽取一张卡
 * @param ctx
 * @param data
 * @return void
 * @throws
 */
public void pickHero(ChannelHandlerContext ctx, ProtoMessage data) {
    // 抽取卡牌的接口, 需要使用 Json-Rpc 去逻辑接口调用逻辑服务器来获取卡牌
    long userid = data.getUserid();
    int num = data.getData().getIntValue("num");// 从卡牌抽卡数量
    JSONArray ret = new JSONArray();
    JsonRpcHttpClient client = null;
    try {
        client = new JsonRpcHttpClient(new URL(getLogicUrl(userid)));
        for (int i = 0; i < num; i++) {
            JSONArray heroJson = client.invoke("pickFightHero",
                userid + "", JSONArray.class);
            ret.add(heroJson);
        }
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (Throwable e) {
        e.printStackTrace();
    }
    SocketHandler.writeJSON(ctx, ret);
}

public void exitScene(long userid) {
    exitWaitingUsers(userid);
    exitPkSceneMap(userid);
    ChannelUser channelUser = ChannelMgr.getInstance().findByUserid(userid);
    if (channelUser != null) {
        ChannelMgr.getInstance().removeChannel(
            ChannelMgr.getInstance().findByUserid(userid).channel);
    }
}

```

```
}

public void exitWaitingUsers(long userid) {
    if (waitingUsers.contains(Long.valueOf(userid))) {
        waitingUsers.remove(Long.valueOf(userid));
        logger.info("玩家" + userid + "已退出等待队列");
    }
}

public void exitPkSceneMap(long userid) {
    if (pkSceneMap.containsKey(Long.valueOf(userid))) {
        ChannelGroup group = pkSceneMap.get(Long.valueOf(userid));
        pkSceneMap.remove(Long.valueOf(userid));
        SocketHandler.writeJSON(group, "玩家" + userid + "已退出对战场景");
        logger.info("玩家" + userid + "已退出对战场景");
    }
}

/**
 * @Title: getLogicUrl
 * @Description: 获取逻辑地址
 * @param userid
 * @return
 * @return String
 * @throws
 */
public String getLogicUrl(long userid) {
    int serverId = (int) (userid % 1000);
    String ip = GameInit.cfg.get("loginServer", "127.0.0.1");
    int port = GameInit.cfg.get("loginPort", 81);
    String url = null;
    JsonRpcHttpClient client = null;
    try {
        client = new JsonRpcHttpClient(new URL("http://" + ip + ":" + port));
        url = client.invoke("getServerUrl", serverId + "", String.class);
    } catch (MalformedURLException e) {
        e.printStackTrace();
        return null;
    } catch (Throwable e) {
        e.printStackTrace();
    }
}
```

```

        return null;
    }
    return url;
}
}

```

【代码解析】游戏中的实时战斗部分主要通过将同一个场景中的玩家放入到同一个 ChannelGroup 进行管理。一个玩家操作的时候，需要将信息通过 ChannelGroup 同步给另一个人，以达到实时效果，但由于网络必定会有不同步的情况出现，所以在客户端与服务器同步的问题上，还需要更加深入的研究，比如对客户端进行预判等，这里不再做深入研究。

## 9.10

## 客户端模拟

以上所有的服务器接口均需要客户端开发对应的交互才能看到实际效果，为了模拟这个实际效果，我们可以直接利用 Java 代码模拟客户端，我使用 Java Swing 做成了可视化的界面形式进行测试，能看到更加直观的数据。

### 9.10.1 登录界面

登录界面包括注册和登录功能。注册时，如果账号不重复，则注册成功，否则失败。登录时，如果密码错误，提示密码错误；如果账号不存在，则提示账号不存在，只有账号和密码都正确才能进入游戏。其完整代码如下：

```

package com.hjc.herolclient;

import java.awt.EventQueue;

public class Login {

    private JFrame frame;
    private JPasswordField pwdText;
    private JTextField nameText;

```



```
private static final String LOGIN_URL = "http://127.0.0.1:81/herolrouter/
route/login";
private static final String REGIST_URL = "http://127.0.0.1:81/herolrouter/
route/regist";
private static final String GET_SERVER_URL = "http://127.0.0.1:81/
herolrouter/route/getServers";

// private static final String LOGIN_REGIST_URL =
// "http://127.0.0.1:81/herolrouter/route/loginOrRegist";

/**
 * Launch the application.
 */
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                Login window = new Login();
                window.frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

/**
 * Create the application.
 */
public Login() {
    initialize();
}

/**
 * Initialize the contents of the frame.
 */
private void initialize() {
    frame = new JFrame();
    frame.setBounds(100, 100, 430, 290);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```



```

frame.getContentPane().setLayout(null);
int w = (Toolkit.getDefaultToolkit().getScreenSize().width - frame
    .getWidth()) / 2;
int h = (Toolkit.getDefaultToolkit().getScreenSize().height - frame
    .getHeight()) / 2;
frame.setLocation(w, h);
JLabel lbldemo = new JLabel("皇室战争 Demo");
lbldemo.setFont(new Font("微软雅黑", Font.BOLD, 18));
lbldemo.setBounds(154, 10, 193, 48);
frame.getContentPane().add(lbldemo);

JLabel label = new JLabel("账号: ");
label.setFont(new Font("微软雅黑", Font.PLAIN, 12));
label.setBounds(100, 81, 64, 17);
frame.getContentPane().add(label);

JLabel label_1 = new JLabel("密码: ");
label_1.setFont(new Font("微软雅黑", Font.PLAIN, 12));
label_1.setBounds(101, 121, 64, 17);
frame.getContentPane().add(label_1);

pwdText = new JPasswordField();
pwdText.addKeyListener(new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
        if (e.getKeyCode() == 10) {
            login();
        }
    }
});
pwdText.setBounds(154, 119, 160, 21);
frame.getContentPane().add(pwdText);

nameText = new JTextField();
nameText.addKeyListener(new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
        if (e.getKeyCode() == 10) {
            login();
        }
    }
});

```

```
    }  
    });  
    nameText.setBounds(154, 79, 160, 21);  
    frame.getContentPane().add(nameText);  
    nameText.setColumns(10);  
  
    JButton registBtn = new JButton("注册");  
    registBtn.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            String name = nameText.getText();  
            String pwd = new String(pwdText.getPassword());  
            if (name.length() == 0) {  
                JOptionPane.showMessageDialog(frame, "请输入账号");  
                return;  
            }  
            if (pwd.length() == 0) {  
                JOptionPane.showMessageDialog(frame, "请输入密码");  
                return;  
            }  
            try {  
                String ret = HttpClient.post(REGIST_URL, "name=" + name  
                    + "&pwd=" + pwd + "");  
                JSONArray jsonRet = JSONArray.parseArray(ret);  
                String msg = jsonRet.getString(1);  
                JOptionPane.showMessageDialog(frame, msg);  
            } catch (Exception e1) {  
                JOptionPane.showMessageDialog(frame, "服务器繁忙, 请稍后  
再试");  
            }  
        }  
    });  
    registBtn.setFont(new Font("微软雅黑", Font.PLAIN, 12));  
    registBtn.setBounds(100, 173, 93, 23);  
    frame.getContentPane().add(registBtn);  
  
    JButton loginBtn = new JButton("登录");  
    loginBtn.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent e) {  
            login();  
        }  
    });
```

```

    });
    loginBtn.setFont(new Font("微软雅黑", Font.PLAIN, 12));
    loginBtn.setBounds(221, 173, 93, 23);
    frame.getContentPane().add(loginBtn);
}

private void login() {
    String name = nameText.getText();
    String pwd = new String(pwdText.getPassword());
    if (name.length() == 0) {
        JOptionPane.showMessageDialog(frame, "请输入账号");
        return;
    }
    if (pwd.length() == 0) {
        JOptionPane.showMessageDialog(frame, "请输入密码");
        return;
    }
    try {
        String ret = HttpClient.post(LOGIN_URL, "name=" + name + "&pwd="
            + pwd + "");
        JSONArray jsonRet = JSONArray.parseArray(ret);
        int code = jsonRet.getIntValue(0);
        String msg = jsonRet.getString(1);
        if (code == 0) {
            // 登录成功
            frame.setVisible(false);
            String servers = HttpClient.post(GET_SERVER_URL, "");
            long userid = jsonRet.getLongValue(2);
            ChooseServer.main(new String[] { servers, userid + "" });
            frame.dispose();
        } else {
            JOptionPane.showMessageDialog(frame, msg);
        }
    } catch (Exception e2) {
        JOptionPane.showMessageDialog(frame, "服务器繁忙, 请稍后再试");
    }
}
}

```

**【运行效果】**登录界面只需要与登录服务器进行交互即可, 其界面如图 9-5 所示。





图 9-5 登录界面

**【代码解析】**通过 Java Swing 构建登录界面，单击“注册”和“登录”按钮分别对服务器的登录服务器发出注册和登录的 HTTP 请求，并获取响应中的 userid。

### 9.10.2 选服界面

选服界面是登录游戏之后选择具体游戏服务器的地方，在这里可以选择要进入的游戏服务器。进入选服界面之后，可以选择服务器，并进入到服务器的主逻辑场景。

代码如下：

```
package com.hjc.herolclient;

import java.awt.EventQueue;

public class ChooseServer {

    public JFrame frame;
    private static JSONArray serversArray;
    private JList<String> list;
    private static long userid;
    public static Map<Integer, JSONArray> serverConfigMap = new HashMap<Integer, JSONArray>();

    /**
     * Launch the application.
     */
    public static void main(String[] args) {
        if (args.length > 0) {
```



```

        serversArray = JSONArray.parseArray(args[0]);
        userid = Long.valueOf(args[1]);
    }
    for (int i = 0; i < serversArray.size(); i++) {
        serverConfigMap.put(i, serversArray.getJSONArray(i));
    }
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                ChooseServer window = new ChooseServer();
                window.frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

/**
 * Create the application.
 */
public ChooseServer() {
    initialize();
}

/**
 * Initialize the contents of the frame.
 */
private void initialize() {
    frame = new JFrame();
    frame.setBounds(100, 100, 430, 290);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(null);

    int w = (Toolkit.getDefaultToolkit().getScreenSize().width - frame
        .getWidth()) / 2;
    int h = (Toolkit.getDefaultToolkit().getScreenSize().height - frame
        .getHeight()) / 2;
    frame.setLocation(w, h);

```

【运行结果】服务器选择界面如图9-6所示。

```
JLabel label = new JLabel("选择服务器");
label.setFont(new Font("微软雅黑", Font.BOLD, 18));
label.setBounds(161, 10, 193, 48);
frame.getContentPane().add(label);

JButton button = new JButton("进入游戏");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        enterGame();
    }
});
button.setFont(new Font("微软雅黑", Font.PLAIN, 12));
button.setBounds(286, 223, 122, 23);
frame.getContentPane().add(button);

JScrollPane scrollPane = new JScrollPane();
scrollPane.setBounds(10, 74, 398, 139);
frame.getContentPane().add(scrollPane);

Vector<String> servers = new Vector<String>();
for (Integer serverId : serverConfigMap.keySet()) {
    JSONArray server = serverConfigMap.get(serverId);
    String serverName = server.getString(3);
    String state = null;
    /* 服务器状态: 0-新服, 1-空闲, 2-繁忙, 3-爆满, 4-维护 */
    switch (server.getIntValue(2)) {
        case 0:
            state = "新服";
            break;
        case 1:
            state = "空闲";
            break;
        case 2:
            state = "繁忙";
            break;
        case 3:
            state = "爆满";
            break;
        case 4:
            state = "维护";
```

```

        break;
    }
    servers.add((serverId + 1) + " " + state + " " + " " + serverName);
}
list = new JList(servers);
list.addKeyListener(new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
        if (e.getKeyCode() == 10) {
            enterGame();
        }
    }
});
scrollPane.setViewportView(list);

JButton button_1 = new JButton("返回");
button_1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        frame.setVisible(false);
        Login.main(null);
    }
});
button_1.setFont(new Font("微软雅黑", Font.PLAIN, 12));
button_1.setBounds(154, 223, 122, 23);
frame.getContentPane().add(button_1);
}

private void enterGame() {
    int index = list.getSelectedIndex();
    if (index == -1) {
        JOptionPane.showMessageDialog(frame, "请选择服务器");
        return;
    }
    Logical.main(new String[] { serverConfigMap.get(index).getString(0),
        serverConfigMap.get(index).getString(1),
        (userid * 1000 + (index + 1)) + " " });
    frame.dispose();
}
}
}

```

【运行结果】服务器选择界面如图 9-6 所示。



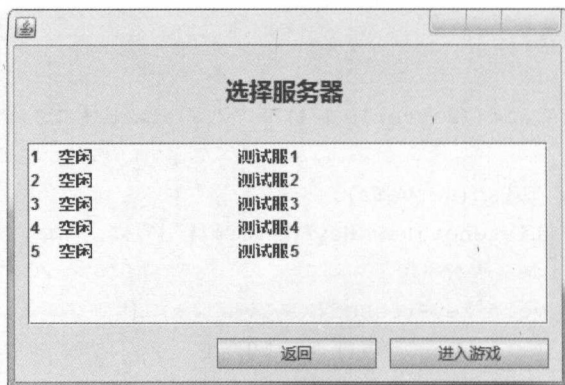


图 9-6 选服界面

**【代码解析】**选服界面主要由 Jlist 列表和“进入游戏”与“返回”两个按钮组成，在 Jlist 列表点击选择的服务器之后进入游戏，就会向登录服务器发出选服的 HTTP 请求，并得到响应，获取其中的服务器 ID 即可进入到游戏主逻辑界面。

### 9.10.3 主逻辑界面

进入主逻辑界面之后，就可以进行所有的主逻辑操作，单击“发送”按钮即可发送请求到逻辑服务器，服务器收到请求并进行处理后，会返回消息到响应消息的窗口。

通过主逻辑界面进入到对战界面，需要先获取对战服务器，获取到对战服务器的地址之后，单击“匹配对战”按钮即可进入到对战界面。

其完整代码如下：

```
package com.hjc.herolclient;

import java.awt.EventQueue;

public class Logical {

    private JFrame frame;
    private static JTextArea reqText;
    private static JTextArea consoleArea;
    private static String LOGIC_URL;
    public static String ip;
    public static int port;
```



```

public static String pvpIp;
public static int pvpPort;
private static long userid;

/**
 * Launch the application.
 */
public static void main(String[] args) {
    ip = args[0];
    port = Integer.parseInt(args[1]);
    userid = Long.parseLong(args[2]);
    LOGIC_URL = "http://" + ip + ":" + port + "";
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                Logical window = new Logical();
                window.frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

/**
 * Create the application.
 */
public Logical() {
    initialize();
}

/**
 * Initialize the contents of the frame.
 */
private void initialize() {
    frame = new JFrame();
    frame.setBounds(100, 100, 570, 530);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().setLayout(null);
}

```

```
int w = (Toolkit.getDefaultToolkit().getScreenSize().width - frame
        .getWidth()) / 2;
int h = (Toolkit.getDefaultToolkit().getScreenSize().height - frame
        .getHeight()) / 2;
frame.setLocation(w, h);

JLabel label = new JLabel("逻辑场景");
label.setFont(new Font("微软雅黑", Font.BOLD, 18));
label.setBounds(230, 10, 187, 46);
frame.getContentPane().add(label);

JLabel label_1 = new JLabel("请求消息: ");
label_1.setFont(new Font("微软雅黑", Font.PLAIN, 12));
label_1.setBounds(10, 84, 182, 15);
frame.getContentPane().add(label_1);

JScrollPane scrollPane = new JScrollPane();
scrollPane.setBounds(10, 109, 543, 116);
frame.getContentPane().add(scrollPane);

JButton button = new JButton("发送");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        sendReq();
    }
});

button.setFont(new Font("微软雅黑", Font.PLAIN, 12));
scrollPane.setColumnHeaderView(button);

reqText = new JTextArea();
reqText.setText("{\"typeid\":\"" + PvpProtoIds.TEST
        + "\", \"data\":{\"}, \"userid\":\"" + userid + "\"}");
scrollPane.setViewportViewView(reqText);

JLabel label_2 = new JLabel("响应消息: ");
label_2.setFont(new Font("微软雅黑", Font.PLAIN, 12));
label_2.setBounds(10, 242, 182, 15);
frame.getContentPane().add(label_2);
```

```

JScrollPane scrollPane_1 = new JScrollPane();
scrollPane_1.setBounds(10, 267, 543, 219);
frame.getContentPane().add(scrollPane_1);

consoleArea = new JTextArea();
scrollPane_1.setViewportViewView(consoleArea);

JButton button_1 = new JButton("清空");
button_1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        consoleArea.setText("");
    }
});
button_1.setFont(new Font("微软雅黑", Font.PLAIN, 12));
scrollPane_1.setColumnHeaderView(button_1);

JButton button_2 = new JButton("匹配对战");
button_2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // 进入匹配对战服务器
        if (pvpIp == null || pvpPort == 0) {
            JOptionPane.showMessageDialog(frame, "请先获取对战服务器
信息");

            return;
        }
        Pvp.main(new String[] { pvpIp, pvpPort + "", ip, port + "",
            userid + "" });
        frame.dispose();
    }
});
button_2.setFont(new Font("微软雅黑", Font.PLAIN, 12));
button_2.setBounds(460, 80, 93, 23);
frame.getContentPane().add(button_2);

JButton button_3 = new JButton("获取对战服务器");
button_3.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            String ret = HttpClient.post(LOGIC_URL, "data={\"typeid\":\"
+ LogicProtoIds.GET_PVP_SERVER

```



```

        + "\", \"data\": {}, \"userid\": \" + userid + \"\"");
        // 获取 PVP 服务器
        pvpIp = JSONArray.parseArray(ret).getString(0);
        pvpPort = JSONArray.parseArray(ret).getIntValue(1);
    } catch (Exception e1) {
        e1.printStackTrace();
    }
}

});
button_3.setFont(new Font("微软雅黑", Font.PLAIN, 12));
button_3.setBounds(269, 80, 176, 23);
frame.getContentPane().add(button_3);
}

private void sendReq() {
    String sendMsg = reqText.getText();
    try {
        JSON.parseObject(sendMsg, ProtoMessage.class);
    } catch (Exception e) {
        JOptionPane.showMessageDialog(frame, "请输入规定协议格式的数据");
        return;
    }
    if (reqText.getText().length() == 0) {
        JOptionPane.showMessageDialog(frame, "请输入请求信息");
        return;
    }
    try {
        String ret = HttpClient.post(LOGIC_URL, "data=" + sendMsg);
        consoleArea.append(ret + "\r\n");
    } catch (Exception e1) {
        e1.printStackTrace();
        JOptionPane.showMessageDialog(frame, "服务器繁忙, 请稍后再试");
    }
}
}
}

```

【运行结果】主逻辑界面如图 9-7 所示。



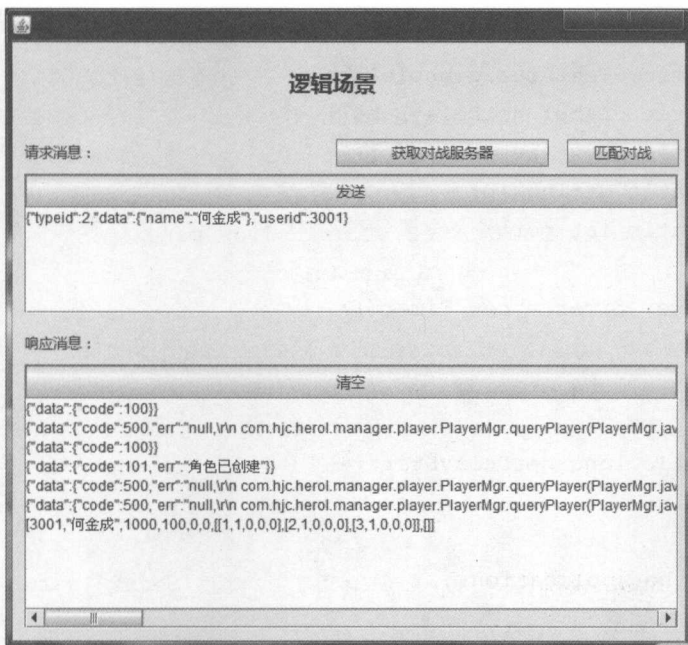


图 9-7 主逻辑界面

**【代码解析】**在逻辑场景界面，可以通过输入请求信息发出 HTTP 请求，并将请求信息显示在响应消息中。通过点击“获取对战服务器”按钮可以对逻辑服务器发出请求，获取对战服务器的地址端口信息，再点击“匹配对战”按钮即可进入实时对战界面。

#### 9.10.4 对战界面

进入到对战界面后，与对战服务器建立 TCP 连接，并每秒发送一个消息测试网络延迟，同主逻辑界面一样，输入请求信息，点击“发送”按钮即可请求对战服务器进行逻辑处理，并获取到返回结果放到响应消息中显示，点击“退出对战”按钮可断开与对战服务器的连接，并返回主逻辑服务器。

对战客户端的代码如下:

```
package com.hjc.herolclient;

import java.awt.EventQueue;

public class Pvp {
```

```
private JFrame frame;
public static JTextArea consoleArea;
public static JLabel netDelayLabel;
public static JTextArea reqText;
private static String ip;
private static int port;

private Timer timer = new Timer();

public static long userid;

public static long netDelayStart;

/**
 * Launch the application.
 */
public static void main(String[] args) {
    final String pvpIp = args[0];
    final int pvpPort = Integer.parseInt(args[1]);
    ip = args[2];
    port = Integer.parseInt(args[3]);
    userid = Long.parseLong(args[4]);
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                Pvp window = new Pvp();
                window.frame.setVisible(true);
                new NettyClient().connect(pvpPort, pvpIp);
                window.netDelayTest();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

public void netDelayTest() {
    timer.schedule(new TimerTask() {
        @Override
        public void run() {
```

```

        // 每秒检查网络延迟
        JSONObject req = new JSONObject();
        req.put("typeid", PvpProtoIds.TEST);
        netDelayStart = System.currentTimeMillis();
        if (SocketInHandler.ctx != null) {
            SocketInHandler.write(SocketInHandler.ctx,
                req.toJSONString());
        } else {
            JOptionPane.showMessageDialog(frame, "对战服务器繁忙, 请
稍后再试");

            Logical.main(new String[] { ip, port + "", userid + "" });
            frame.dispose();
        }
    }

    }, new Date(System.currentTimeMillis() + 1000), 1000);
}

/**
 * Create the application.
 */
public Pvp() {
    initialize();
}

/**
 * Initialize the contents of the frame.
 */
private void initialize() {
    frame = new JFrame();
    frame.addWindowListener(new WindowAdapter() {
        @Override
        public void windowClosed(WindowEvent e) {
            SocketInHandler.ctx.channel().close();
            // 程序优雅退出, 释放 NIO 线程组
            NettyClient.workGroup.shutdownGracefully();
            NettyClient.shutdown();
        }
    });
    frame.setBounds(100, 100, 570, 530);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

```



```
frame.getContentPane().setLayout(null);

int w = (Toolkit.getDefaultToolkit().getScreenSize().width - frame
        .getWidth()) / 2;
int h = (Toolkit.getDefaultToolkit().getScreenSize().height - frame
        .getHeight()) / 2;
frame.setLocation(w, h);

JScrollPane scrollPane = new JScrollPane();
scrollPane.setBounds(10, 267, 543, 219);
frame.getContentPane().add(scrollPane);

JButton button_1 = new JButton("清空");
button_1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        consoleArea.setText("");
    }
});
button_1.setFont(new Font("微软雅黑", Font.PLAIN, 12));
scrollPane.setColumnHeaderView(button_1);

consoleArea = new JTextArea();
scrollPane.setViewportViewView(consoleArea);

JLabel label = new JLabel("延迟: ");
label.setBounds(446, 14, 54, 15);
frame.getContentPane().add(label);

netDelayLabel = new JLabel("0ms");
netDelayLabel.setBounds(499, 14, 54, 15);
frame.getContentPane().add(netDelayLabel);

JLabel label_1 = new JLabel("响应消息: ");
label_1.setFont(new Font("微软雅黑", Font.PLAIN, 12));
label_1.setBounds(10, 242, 182, 15);
frame.getContentPane().add(label_1);

JLabel label_2 = new JLabel("实时对战场景");
label_2.setFont(new Font("微软雅黑", Font.BOLD, 16));
label_2.setBounds(207, 14, 187, 46);
```



```

frame.getContentPane().add(label_2);

JScrollPane scrollPane_1 = new JScrollPane();
scrollPane_1.setBounds(10, 109, 543, 116);
frame.getContentPane().add(scrollPane_1);

JButton button = new JButton("发送");
scrollPane_1.setColumnHeaderView(button);
button.setFont(new Font("微软雅黑", Font.PLAIN, 12));

reqText = new JTextArea();
scrollPane_1.setViewportViewView(reqText);

reqText.setText("{\"typeid\":\"" + PvpProtoIds.TEST
    + "\",\"data\":{\"},\"userid\":\"" + userid + "\"}");

button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        sendReq();
    }
});

JLabel label_3 = new JLabel("请求消息: ");
label_3.setFont(new Font("微软雅黑", Font.PLAIN, 12));
label_3.setBounds(10, 84, 182, 15);
frame.getContentPane().add(label_3);

JButton button_2 = new JButton("退出对战");
button_2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Logical.main(new String[] { ip, port + "", userid + "" });
        SocketInHandler.write(SocketInHandler.ctx, "{\"typeid\":\""
            + PvpProtoIds.EXIT_SCENE + "\",\"data\":{\"},\"userid\":\""
            + userid + "\"}");
        NettyClient.workGroup.shutdownGracefully();
        timer.cancel();
        frame.dispose();
    }
});
button_2.setFont(new Font("微软雅黑", Font.PLAIN, 12));

```

```

        button_2.setBounds(460, 76, 93, 23);
        frame.getContentPane().add(button_2);
    }

    private void sendReq() {
        String sendMsg = reqText.getText();
        try {
            JSON.parseObject(sendMsg, ProtoMessage.class);
        } catch (Exception e) {
            JOptionPane.showMessageDialog(frame, "请输入规定协议格式的数据");
            return;
        }
        if (reqText.getText().length() == 0) {
            JOptionPane.showMessageDialog(frame, "请输入请求信息");
            return;
        }
        SocketInHandler.write(SocketInHandler.ctx, reqText.getText());
    }
}

```

其使用 Netty 作为 TCP 客户端，Netty 部分代码如下：

```

package com.hjc.net;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;

public class NettyClient {
    public static NioEventLoopGroup workGroup;
    private static ChannelFuture futrue;

    public void connect(final int port, final String host) throws Exception {
        workGroup = new NioEventLoopGroup(); // work 线程组
        // 配置客户端 NIO 线程组
    }
}

```

```

Bootstrap b = new Bootstrap();
b.group(workGroup).channel(NioSocketChannel.class)
    .option(ChannelOption.TCP_NODELAY, true)
    .handler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch) throws
Exception {
            ch.pipeline().addLast(new StringDecoder());
            ch.pipeline().addLast(new StringEncoder());
            ch.pipeline().addLast(new SocketInHandler());
        }
    });
// 绑定端口, 同步等待
futtrue = b.connect(host, port);
}

public static void shutdown() {
    // 等待服务监听端口关闭
    try {
        futtrue.channel().closeFuture().sync();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws Exception {
    int port = 8700;
    new NettyClient().connect(port, "localhost");
}
}

```

处理器代码如下:

```

package com.hjc.net;

import io.netty.channel.ChannelHandlerAdapter;
import io.netty.channel.ChannelHandlerContext;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

```



```
import com.alibaba.fastjson.JSONObject;
import com.hjc.herolclient.Pvp;

public class SocketInHandler extends ChannelHandlerAdapter {
    public static ChannelHandlerContext ctx = null;

    private static final Logger logger = LoggerFactory
        .getLogger(SocketInHandler.class);

    public static void write(ChannelHandlerContext ctx, String msg) {
        ctx.writeAndFlush(msg);
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        SocketInHandler.ctx = ctx;
        JSONObject req = new JSONObject();
        req.put("typeid", PvpProtoIds.TEST);
        Pvp.netDelayStart = System.currentTimeMillis();
        SocketInHandler.write(SocketInHandler.ctx, req.toJSONString());
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
        throws Exception {
        String body = (String) msg;
        if (body.equals("delay")) { // 测试网络延迟
            long delay = System.currentTimeMillis() - Pvp.netDelayStart;
            Pvp.netDelayLabel.setText(delay + "ms");
        } else {
            System.out.println("client read: " + body);
            Pvp.consoleArea.append(body + "\r\n");
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        // 释放资源
        logger.warn("Unexpected exception : " + cause.getMessage());
        ctx.close();
    }
}
```



```
}  
}  
}
```

【运行结果】图 9-8 和图 9-9 为同一个对战场景中的两个模拟客户端。

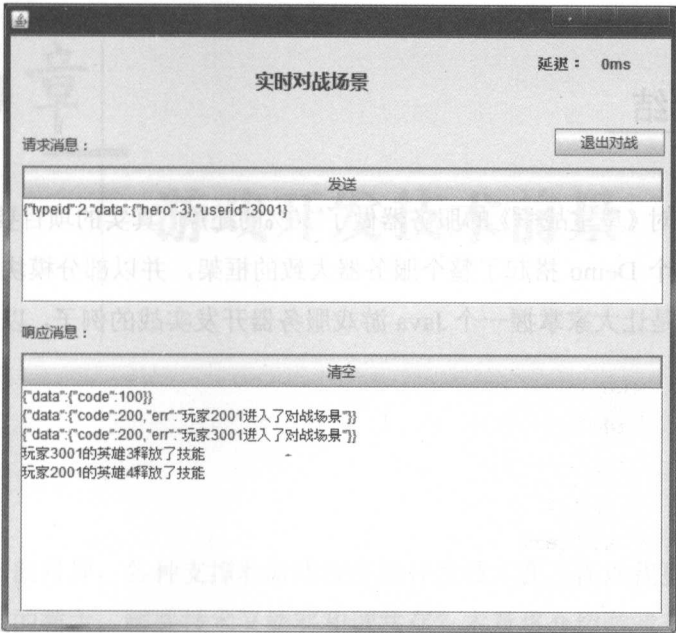


图 9-8 对战场景客户端 (1)

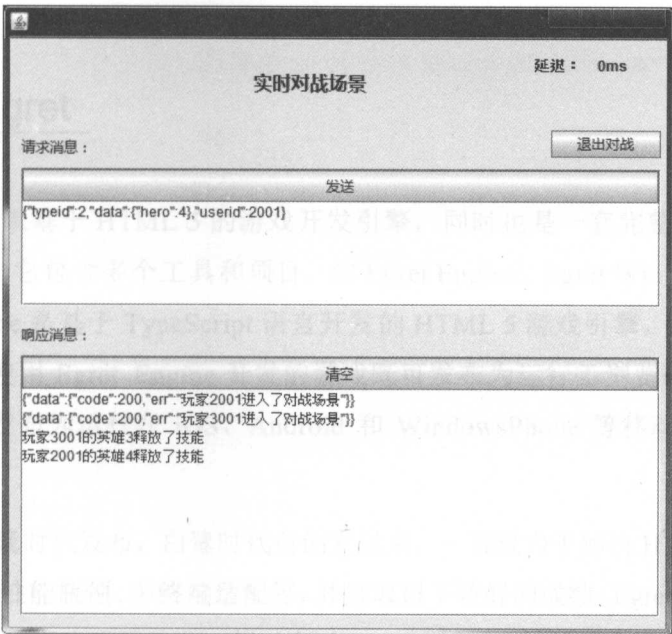


图 9-9 对战场景客户端 (2)

**【代码解析】**对战场景界面同主逻辑界面一样，输入请求信息并点击“发送”按钮之后，即可发送消息到服务端，玩家的攻击动作、释放技能动作、移动动作等都可以通过在界面中发送消息进行模拟。

## 9.11 总结

本章内容只对《皇室战争》的服务器做了 30% 的还原，真实的项目要比介绍的 Demo 庞大很多，但这个 Demo 搭起了整个服务器大致的框架，并以部分模块为例进行了实战开发。本章主要是让大家掌握一个 Java 游戏服务器开发实战的例子，以了解开发流程和思路。

## 第10章

# 游戏开发技术前景

游戏行业日新月异，各种支撑着游戏程序的开发技术在不断迭代更新，哪些技术能成为某一个领域的霸主，哪些技术又能够和谐共存？本章将介绍游戏开发中各种开发技术（不限客户端技术或服务端技术）及其前景。

### 10.1 Egret

Egret 是一款基于 HTML 5 的游戏开发引擎，同时也是一套完整的 HTML 5 游戏开发解决方案。它包含多个工具和项目，如 Egret Engine、Egret Wing、Lark、EgretVS 等。Egret Engine 是基于 TypeScript 语言开发的 HTML 5 游戏引擎，该项目在 BSD 许可证下发布。使用 Egret Engine 开发的游戏既可发布为运行于浏览器之中的 HTML5 版本，也可以发布为运行在 iOS、Android 和 WindowsPhone 等移动终端操作系统的原生程序。

Egret 由白鹭时代发布，白鹭时代自创立以来，一直致力于解决 HTML5 游戏开发中的各种问题，如性能瓶颈、多终端适配等，并且取得了较好的成绩。Egret 支持 TypeScript、JavaScript 和 ES6 语言进行开发，相对来说，Egret 对 Flash 开发者更为友好，早年间盛

极一时的 Flash 游戏的开发者可以通过 Egret 快速转到 HTML 5 游戏的开发，这也能使 HTML 5 游戏行业快速发展。

## 10.2

### Cocos 2D

Cocos 家族的第一个产品是 2005 年由阿根廷一位热爱游戏开发的人创作的，第一个版本是用 Python 语言编写的。第一版引擎诞生之后，乔布斯发现其商机，并与他合作，于是很快就出现了由 Objective-C 编写的 Cocos 2D for iPhone 0.1 版本，它与 Python 的设计思路如出一辙。这款引擎一诞生，iPhone 上的游戏便多了起来，截至 2008 年 12 月，App Store 已经有超过 40 个用 Cocos 2D 引擎开发的游戏。自此以后，Cocos 2D 各个平台的各种语言版本开始涌现，如 ShinyCocos（Ruby bindings）、Cocos 2D-Android（Java based）、cocosNet（Mono based）。

各个平台各种语言的版本越来越多，Cocos 2D-X（X 代表 Cross）的出现使这个版本的 Cocos 2D 引擎具有划时代的意义。Cocos 2D-X 为开发者提供了跨平台支持，通过 C++ 写出游戏逻辑之后即可编译打包在 iOS、Android 及其他平台上运行。

现在，Cocos 已经涌现了非常多的版本，如下。

- （1）ShinyCocos: Cocos 2D-iphone 绑定 Ruby 的实现。
- （2）CocosNet: Cocos 2D 的 .Net 实现，运行在 Mono 上。
- （3）Cocos 2D-Android: Cocos 2D 的 Java 实现，运行在 Android 操作系统。
- （4）Cocos 2D-Android1: 中国某资深开发者对另一个关于 Android 操作系统的 Cocos 2D 引擎的实现。
- （5）Cocos 2D-A: Cocos 2D 的 C++ 语言移植版。
- （6）Cocos 2D-XNA: 某创业团队成员实现的支持 Windows Phone 7.0 版 Cocos 2D。
- （7）Cocos 2D-HTML5: 由 Cocos 2D-X 团队开发的分支，支持 HTML5 Canvas 技术，获得 Google 等公司的支持。



(8) Cocos 2D-JavaScript: Cocos 2D 的 JavaScript 语言实现, 结合 HTML 5 技术的发展, Cocos 2D-JavaScript 会有更广阔的前景。

(9) Cocos 2D-X for WP8 2.0: 利用 Direct 3D 11 的 Cocos 2D 实现。

(10) Cocos 2D-lua: Cocos 2D 的 Lua 语言实现, 实现游戏逻辑的脚本化。

Cocos 除了涌现如此之多的版本之外, 还出现了很多一整套的开发流程工具, 如 Cocos 商店、Cocos Builder、Cocos IDE 等, 这些工具 Cocos 开发高效稳定、有更短的开发周期、更好的游戏质量。

为顺应现在游戏的发展潮流, Cocos 除了称霸 2D 游戏市场之外, 也推出了 Cocos 的 3D 版本, 使使用 Cocos 开发 3D 游戏或 VR 游戏都不再是难事。

## 10.3

### Unity

Unity 也叫作 Unity 3D, 是一套完整的游戏引擎, 包括了图形编辑、声音编辑、物理模拟等功能, 并且提供了强大的关卡编辑器, 在 3D 软件格式的支持上也非常不错, 能支持大部分主流 3D 软件格式, Unity 可以使用 C# 或 JavaScript 等高级语言实现脚本功能。在 Unity 开发中, 开发者无须了解底层复杂的技术, 就能快速开发出具有高性能、高品质的游戏产品。

Unity 是跨平台的 3D 游戏引擎, 支持的平台包括 PC、Mac、Linux、Web、iOS、Android、Xbox 360、Play Station 3 等大部分主流游戏平台, 还可以将游戏直接导出为 Flash 格式放到网页上, 很多时候, 可以选择在 PC 平台开发和测试, 只需要很少的改动即可将游戏移植到其他平台。

Unity 是一个成熟的游戏引擎, 有强大的功能、良好的可移植性, 随着 iOS、Android 手机的大量普及和 3D 网页游戏的兴起, Unity 在手机和网页平台有着广泛的应用和传播。在手机移动市场上可以找到大量使用 Unity 开发的游戏, 如 Battle Heart、Zombievillage USA 等。Unity 不仅能开发 3D 游戏, 开发 2D 游戏更是不在话下。

除此之外, Unity 还内置了网络引擎 Raknet, 如果只需要完成玩家与玩家之间直接连接的网络游戏, 这些功能已经足够。

在 3D 游戏和 VR 兴起的时代,选择 Unity 作为客户端开发技术也是一个不错的选择,除了完整的解决方案,引擎本身的性能也是非常可观的。

## 10.4 Unreal

Unreal (虚幻引擎)的全称是 Unreal Engine,是目前最顶尖的游戏引擎,占有全球游戏市场 80%的份额。

基于 Unreal 开发的大作无数,如《战争机器》、《质量效应》、《生化奇兵》、TERA、《战地之王》、《剑灵》、《无尽之刃》等。相信以上大作大家都听说过,也见识过其渲染后的精美画面。

Unreal Engine 3D 引擎目前采用最新的一些技术,包括即时光迹追踪、虚拟位移、HDR 光照技术等,并且 Unreal 每秒运算的多边形个数能达到两亿多个。Unreal Engine 3D 引擎与 NVIDIA 的 GeForce 6800 显卡相配合,能够运算出电影 CG 等级的精美画面。

Unreal 曾经是 PC 主机游戏的霸主,相信在未来的技术中,其也将起到不可忽视的作用。

## 10.5 Java

以上几个技术均为游戏的客户端技术,但如果需要做网络游戏,服务器技术也是必不可少的,随着页游和手游市场的兴起,对游戏开发技术的需求越来越多。网络游戏开发是一个非常庞大的体系,总体分为客户端与服务端,客户端是在玩家面前展示的游戏表达,服务器端是处理游戏运行中逻辑与数据的部分。由于一台服务器需要支持众多玩家的请求,所以服务器的性能决定了游戏玩家同时在线数量。

Java 作为服务端开发语言,是可以胜任游戏服务器的需求的,Java 的很多特性非常适合为游戏服务器服务。

在 Java 虚拟机(JVM)诞生后的 20 年中,不少系统都部署了 JVM, JVM 也经历了

无数次的漏洞修复和性能提升。JVM 的优点有以下几个方面。

(1) JVM 完美支持日志和监控，可以很方便地监控小到单个线程的性能指标。

(2) JVM 有优化的垃圾回收器，有非常友好的垃圾回收算法。

(3) Java 的跨平台兼容性也得益于 JVM。只需要一个 JVM，Java 就能运行起来。

Java 至今仍存在，也要得益于其有大量的第三方库支持，Google、Amazon、Linedln、Twitter 及很多 Apach 项目都倚重于 Java，并且 Java 有大量开源的第三方库。

Java 的特性是完全能够满足游戏服务器的需求的，只要使用得当，使用 Java 开发的游戏服务器一定是稳定高效的。

## 10.6 Node.js

Node.js 已经发展得越来越好，使用人数也在不断增加，应用范围更是越来越广。从传统企业应用、互联网的使用，到云计算的发展，随处可见 Node.js 的身影。它之所以这么受欢迎，除了其本身的事件模型及 V8 性能优化等一些特性之外，还与国内外很多开源项目有关，如网易、淘宝等的带头使用。采用 Node.js 实现高性能和可扩展性的游戏服务器将是一件非常有意义的工作。

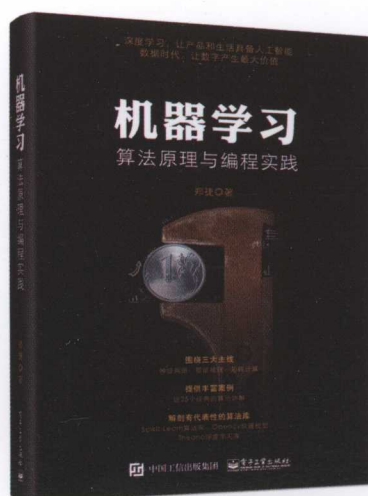
使用 Node.js 开发网络游戏服务器，可以充分利用 JavaScript 动态脚本语言的性质、GoogleV8 引擎的优越性能，它还在网络数据传输的异步化方面具有快速、实时等事件的编程模型，同时，它还有着强大的开源社区。Node.js 在 HTML 5 的应用开发中，具有前后端代码共享、DSL 灵活定义等特点。但 Node.js 也有 GoogleV8 在内存使用中存在的限制问题，尽管 V8 引擎通过动态编译对脚本语言进行了静态化，但如果大规模开发中忽略了一些代码的细节，比如代码的额外异常检查和类属性的动态变化等，会导致未经过调优的代码的性能不尽如人意，但经过调优的代码还是可以达到静态语言的性能的。

Node.js 的事件驱动模型、单线程等特性，都是很符合游戏服务器的特性的。Node.js 尽管还有一些缺陷，但若使用得当，一定可以在游戏服务器中发挥巨大的作用。

## 10.7 总结

游戏开发的技术层出不穷，开发者应该根据实际情况对技术进行选型，不能盲目跟风，只要能满足目前的需求、能使游戏高效稳定运行、能有较短的开发迭代周期、能最大化创造利益的技术就是我们要选择的技术。





出版社：电子工业出版社

ISBN：978-7-121-27367-4

《机器学习算法原理与编程实践》是机器学习原理和算法编码实现的基础性读物，内容分为两大主线：单个算法的原理讲解和机器学习理论的发展变迁。算法除包含传统的分类、聚类、预测等常用算法之外，还新增了深度学习、贝叶斯网、隐马尔科夫模型等内容。对于每个算法，均包括提出问题、解决策略、数学推导、编码实现、结果评估几部分。数学推导力图做到由浅入深，深入浅出。结构上数学原理与程序代码一一对照，有助于降低学习门槛，加深对公式的理解，起到推广和扩大机器学习的作用。



# 深度解析 Java 游戏服务器开发

本书从游戏服务器实战出发，系统讲解了游戏开发的架构及关键技术。由于时延和同时在线的并发性能对玩家的体验非常重要，因此书中详细介绍了如何基于Netty和Mina构建高性能的游戏通信框架。相信本书能够为从事Java游戏开发的读者提供借鉴和指导。

——华为架构师、《Netty权威指南》作者 李林锋

本书从实际项目需求出发，介绍了游戏服务器端的开发技术，并以商业案例来解析具体游戏服务器的架构，有很大实用价值，适合对游戏感兴趣的开发者阅读。

——深蓝教育CEO 沈大海

随着智能手机的普及，移动游戏的开发成为热门。Java凭借着技术成熟和稳定的特点，成为游戏服务器开发的首选技术。本书深入浅出地带着读者细致学习了服务器开发所需知识，并通过诸多案例详细地分析了各类游戏的架构设计。本书非常适合准备了解或从事服务器研发的开发者学习阅读。

——CSDN知名博主、卓越游戏高级工程师 寒江孤叶

《深度解析Java游戏服务器开发》一书知识体系完善，条理清晰，对目前从事移动游戏行业的开发者有很大帮助，推荐读者阅读本书。

——触控科技VP Jane

随着Java技术逐渐成熟，Java在后端服务器领域的应用越来越广泛，本书介绍了Java在游戏服务器中的应用，由浅入深，从基础到案例，从理论到实践，无论是想初步了解游戏服务器研发，还是要进行更加深入的探索，本书都值得一读。

——京东部门经理 李滨



策划编辑：李 冰

责任编辑：李 冰

封面设计：朝天世纪

ISBN 978-7-121-30142-1



9 787121 301421 >

定价：79.00元